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

Deliverable D3.4
Hybrid Analysis

Date of document: T36

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:
Hybrid Analysis

This document summarises deliverable D3.4 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`.

During Envisage a wide range of formal analyses for the ABS language have been realized, including deadlock detection, deductive verification, resource bound computation, test case generation, type systems. Many of these approaches only unfold their full impact in combination with each other, as well as in combination with the ABS simulation tool. In this report we describe a number of such hybrid approaches that were explored in Envisage Task T3.4 *Hybrid Analysis*. The results go beyond the state-of-art of what is possible in each individual method.

List of Authors

Reiner Hähnle (TUD)
Richard Bubel (TUD)
Eduard Kamburjan (TUD)
Crystal Chang Din (UIO)
Abel García Celestrín (BOL)
Elena Giachino (BOL)
Elvira Albert (UCM)
Miguel Gómez-Zamalloa (UCM)
Miguel Isabel (UCM)
Enrique Martín (UCM)

Contents

1 Introduction **5**

2 Session Types for ABS **7**
2.1 Introduction . . . 7
2.2 Example . . . 8
2.3 Session Types . . . 8
2.4 Verification . . . 10
2.5 Conclusion . . . 11

3 Deadlock Analysis for ABS **12**
3.1 Introduction . . . 12
3.2 The `DF4ABS` Tool . . . 13
3.3 Assessments . . . 13
3.4 Conclusion . . . 14

4 Contract-based Verification of Asynchronous Method Calls with Cooperative Scheduling **15**
4.1 Invariant-based Verification with Cooperative Scheduling . . . 15
4.2 Contract-based Verification . . . 16
4.3 Contract-based Verification with Cooperative Scheduling . . . 18
4.4 Open and Closed World Verification . . . 21
4.5 A Calculus for Contract-Based Verification of Asynchronous Calls . . . 22
4.6 Related and Future Work . . . 25
4.6.1 Related Work . . . 25
4.6.2 Tracking of Active Futures . . . 25
4.6.3 Block Contracts . . . 25
4.6.4 Assumptions on the Scheduler . . . 26

5 Combining Static Analysis and Testing for Deadlock Detection **27**
5.1 Introduction . . . 27
5.2 Motivating Example . . . 27
5.3 Enhanced Semantics with Interleavings Table . . . 28
5.4 Combining Static Deadlock Analysis and Testing . . . 29
5.5 Experimental Evaluation . . . 30
5.6 Conclusions . . . 32

6 Combination of Resource Analysis and Profiling **33**

Bibliography **35**

A Session Types for ABS **39**

B Combining Static Analysis and Testing for Deadlock Detection **56**

Chapter 1

Introduction

According to the Envisage Description of Work, Deliverable 3.4 is concerned with combining different—and complementary—analysis techniques developed during Envisage. This might involve static as well as dynamic (runtime) approaches.

In Chapter 2 we study the application of session type systems [31] to compositional verification of actor-based systems using futures [34]. Session types are extended to specify the communication order between ABS objects. We developed a function to translate session types to *history*-based contracts, which are class invariants for ABS programs and can be verified by the deductive verification tool KeY-ABS [20].

In Chapter 3 we report the latest developments on a type-based approach for deadlock analysis which combines a number of different techniques. This updates the report given in [16, Chapter 5]. One of the strongest points of the presented tool is the flexibility provided by its modular design, although the tool has been initially implemented to target ABS we have been able to adapt it to other languages. A very successful example is the deadlock analysis for JVLM (the *Java Bytecode*), which is currently under development. The inference process in this case is highly complex, for this reason we intend to combine it with KeY to produce more precise behavioural types.

In Chapter 4 we extend the invariant-based approach to deductive verification of ABS programs with unbounded size of input data reported first in [16, Chapters 2–4]. While the latter is situated at the class level, i.e. invariants are proven class-wise, for practical purposes it is often necessary to specify the behavior of each method separately. To this end we generalize the contract-based approach established for verification of sequential program [3] to the ABS setting. To the best of our knowledge, this is the first modular verification method for contract-based verification of concurrent programs. A publication based on the chapter is in preparation.

In Chapter 5 we propose a seamless combination of static analysis and testing for effective deadlock detection as follows: first, from the deadlock analysis of SACO [24] one obtains abstract descriptions of potential deadlock cycles. These are subsequently used to guide the SYCO [7] testing tool (described in deliverable D3.5 [18]) to find associated deadlock traces (or discard them). In this chapter, we use the analysis of SACO rather than the one in Chapter 3, mainly because of the easiest integration with the SYCO tool. However, it should be feasible to use the analysis results provided by the deadlock analyzer in Chapter 3 as well.

In Chapter 6 we describe the integration of SACO and the ABS Erlang simulator. To facilitate the inclusion and usage of upper bounds during the simulation of ABS programs, we realized an extension of the SACO analyzer [4] that generates upper bound functions for each ABS method. These functions take the same parameters as the method and return the evaluation of the upper bound with respect to those parameters. Here, we describe how these two techniques can be used in combination in practice.

An overview of the various combinations of analyses described here, together with some that were already realized in previous deliverables, is provided in Fig. 1.1.

Together, the work described here, as well as in the D4.x.3 (where $x \in \{1, 2, 3\}$) and the D4.5 deliverables, gives evidence for the claims that were merely conjectured in the DoW:

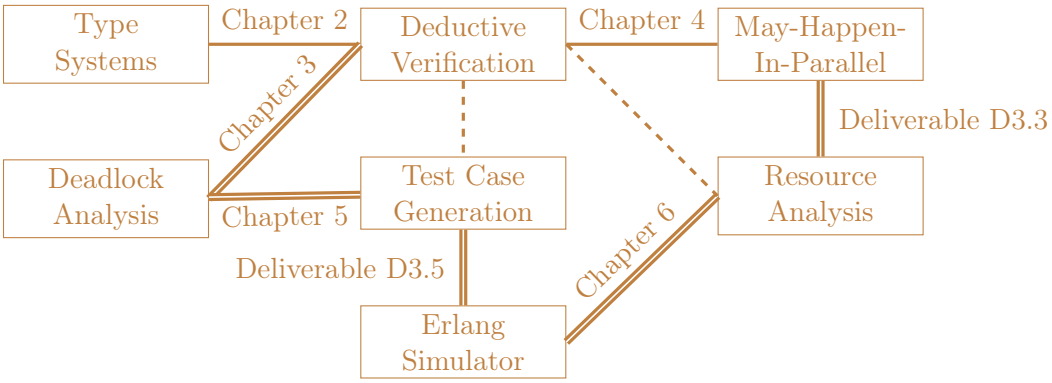


Figure 1.1: Combinations of analysis techniques mentioned in this deliverable. Double lines indicate that the approach is implemented in the Envisage tool suite, dashed lines indicate future work.

1. To formally specify and successfully analyse concurrent, resource-aware industrial systems, it is *necessary* to combine different analysis methods in a concerted manner.

2. The combination yields tools of *sufficient* strength to analyse industrial systems.

In fact, the theme of combining different approaches to software analysis has since the start of Envisage been identified as a major trend in formal verification [8]. The research reported here, therefore, demonstrates that Envisage is situated at the forefront of the state-of-art in formal verification.

List of Papers Comprising Deliverable D3.4

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: Session-Based Compositional Analysis for Actor-Based Languages Using Futures This paper presents a session type system for ABS which is used to verify an ABS system against a specified global communication pattern. The session type system considers the distinguishing features of ABS, such as future-based communication and cooperative scheduling. To capture constraints on scheduling, an automata-based approach to represent and annotate scheduling strategies is proposed.

The paper was written by Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen and will be published in the International Conference on Formal Engineering Methods (ICFEM) 2016.

Paper 2: Combining Static Analysis and Testing for Deadlock Detection This paper introduces a new methodology to find effectively deadlock traces during the systematic testing. The output of the deadlock analysis of SACO is used to generate *deadlock constraints* which must be satisfied during the testing process. Otherwise, the execution is stopped as soon as we are able to prove the trace is deadlock-free. The paper was written by Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel and is published in the International Conference on Integrated Formal Methods (iFM) 2016.

Chapter 2

Session Types for ABS

2.1 Introduction

ABS can verify the behavior of methods and objects with the help of *object invariants* with KeY-ABS. These invariants only specify the state of a single object. When specifing the behavior of a whole system, the global specification needs to be transformed into local specifications for each involved object manually. This is error-prone as there is neither tool support nor a formal foundation for this transformation. This challenge motivates our compositional analysis framework, which models and locally verifies the behaviors of distributed endpoints (i.e. components) from the specification of their global interactions. While our approach does not take concrete values into account, the verified *communication pattern* can serve as a connecting backbone for the local invariants.

We establish a hybrid analysis, which statically type checks local objects' behaviors and, at the same time ensures that local schedulers obey to specified policies during runtime. Our analysis is based on *session types* [31, 43], a static syntax-driven framework which specifies the communication pattern of a system but verifies only the local code of the participants. Session types are an established tool for channel-based communication, mainly π-calculus based systems or languages implementing a similar concurrency model.

To adapt session types for ABS, the distinguishing features of the core ABS concurrency model must be considered: (1) *cooperative scheduling*, where methods explicitly control internal interleavings by explicit scheduling points, and (2) the usage of *futures*, which decouple the process invoking a method, the process executing the method and the process reading the returned value. We have two kinds of communications with future: Caller invoking a method at a remote callee, and callee returning values via a future to those who know that future. The later is non-trivial since several endpoints can read more than once from the same resolved future at any time. As ABS does not use channels, communication between processes is restricted to these two possibilities.

Our approach is to use a *two-fold notion of local types*: Object types define behaviors, including scheduling behavior, among class instances, while method types define behaviors that single processes should follow.

A global specification is projected to several local specification in two step: (1) for each endpoint occurring in it, an object-local type is generated, (2) for each method in an object local type, a method type is generated (3) for each object local type, a *scheduling policy* is generated.

A method type describes the communication of a process (executing a given method) by specifying a set of local communication histories. Our analysis succeeds if every execution is guaranteed to generate a history that is an element of this set.

An object local type describes the communication of an object by specifying a set of sequences of global communication histories. Our analysis succeeds if every execution is guaranteed to generate a history that is a *permutation* of an element of this set, such that for each object its local history within the permutation are indistinguishable from local history in the specified sequence. This expresses that while the system may execute events in a different order than specified, the difference is not visible to any object.

Scheduling policies are represented as *session automata* [10], which specify the correct order of process

activations and reactivations. A policy is added as an annotation when creating an object and is statically checked against the policy derived from the global specification. This approach to scheduling is not related to the real-time scheduling for ABS introduced in [9]. The automata are able to store futures to decide on reactivations, as processes can be identified by the future they are computing, and are able to reject any (re-)activation and let the object wait for a call to arrive.

2.2 Example

Consider a service, called *grading system*, which offers an expensive computation on sensitive data, e.g. automatic evaluation of exams. This service consists of three endpoints: A computation server, denoted by c, and a service desk, denoted by d, where a student, denoted by s, can request his grades. The protocol is as follows: Once c finishes calculating the grades, it sends a *publish* message containing the grades to d and an announcement, *announce* message, to a student. It is not desirable that d starts a new communication with c because c may be already computing the next exams; it is also not desirable that d communicates to s without any request from s. As a last restriction, it is not desirable that d synchronizes its two processes via a field. The service desk has only limited capacity and can not keep track of all wrongly timed requests of students and check after each publishing whether some students waits for these grades - all its processes must be atomic.

If a student requests his/her grades before the service desk receives the grades from c by *publish*, the scheduler of d must postpone the process of *request* until *publish* has been executed and terminated. This is not possible for core ABS, because a scheduler in core ABS cannot be idle while waiting for a specific message when the process queue is non-empty. Thus we propose an extension of core ABS to ensure that the endpoints and their local schedulers behave well to the specified communication order.

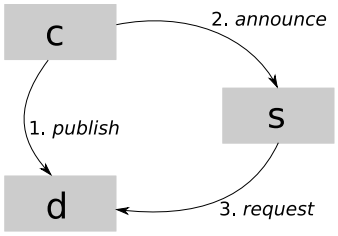


Figure 2.1: A grading system

2.3 Session Types

Global Types

Global types, denoted by $\mathbf{G}$, define global communication specifications within a closed system of objects. Contrary to session types for the π-calculus [43, 31], we do not specify the datatype of a message since the message is a method call or a method return and every method in ABS has a fixed signature. The syntax of $\mathbf{G}$ is defined as follows:

Definition 2.3.1 Let $\mathbf{p}, \mathbf{q}$ range over objects, denoted by Ob, f over futures, m over method names and $\mathtt{C}$ over all constructors of all abstract datatypes.

$$\mathbf{G} \;::=\; \mathbf{0} \xrightarrow{f} \mathbf{q}{:}m \mid \mathbf{G}.\mathbf{g} \qquad \mathbf{g} \;::=\; \mathbf{p} \xrightarrow{f} \mathbf{q}{:}m.\mathbf{g} \mid \mathbf{p}{\downarrow}f{:}(\mathtt{C}).\mathbf{g} \mid \mathbf{p}{\uparrow}f{:}(\mathtt{C}).\mathbf{g} \mid \mathsf{Rel}(\mathbf{p}, f).\mathbf{g} \mid \mathbf{p}\{\mathbf{g}_j\}_{j\in J} \mid \mathsf{end} \mid \mathbf{g}^*$$

Initialization $\mathbf{0} \xrightarrow{f} \mathbf{q}:m$ starts interactions from the *main block* invoking object $\mathbf{q}$. We use . for sequential composition and write $\mathbf{G}.\mathbf{g}$ to mean interaction(s) $\mathbf{g}$ follows $\mathbf{G}$. Interaction $\mathbf{p} \xrightarrow{f} \mathbf{q}:m$ models a remote call, where object $\mathbf{p}$ asynchronously calls method m at object $\mathbf{q}$ via future f, and then $\mathbf{q}$ creates a new process for this method call. The resolving type $\mathbf{p}{\downarrow}f:(\mathtt{C})$ models object $\mathbf{p}$ resolving the future f. If the method has an algebraic datatype as its return type, then the return value has C as its outermost constructor; otherwise we simply write $\mathbf{p}{\downarrow}f$. The fetching type $\mathbf{p}{\uparrow}f:(\mathtt{C})$ models object $\mathbf{p}$ reading the future f. The usage of C here is similar to the one in $\mathbf{p}{\downarrow}f:(\mathtt{C})$. The releasing type $\mathsf{Rel}(\mathbf{p}, f)$ models $\mathbf{p}$ which releases the control until future f has been resolved. This type corresponds to await f? statement in *ABS*.

The branching type $\mathbf{p}\{\mathbf{g}_j\}_{j\in J}$ expresses that as $\mathbf{p}$ selects the jth branch, $\mathbf{g}_j$ guides the continuing interactions. The type end means termination. The type $\mathbf{g}^*$ describes repetition. Note that only a *self-contained* $\mathbf{g}$ can be repeatedly used. We say $\mathbf{g}$ is *self-contained* if (1) wherever there is a remote call or releasing, there is a corresponding resolving and visa versa; and (2) it contains no end, and (3) every repeated type within it is also *self-contained*. In other words, we need to keep *linearity* of futures for every iteration.

Example 1 *The global type* grades *specifies the grading system discussed in Section 2.2:*

$$\begin{aligned}\mathsf{grades} = \ \mathbf{0} \xrightarrow{f_0} \mathbf{c}{:}\mathit{pubGrd}.\mathbf{c} \xrightarrow{f} \mathbf{d}{:}\mathit{publish}.\mathbf{d}{\downarrow}f.\mathbf{c} \xrightarrow{f'} \mathbf{s}{:}\mathit{announce}.\\ \mathbf{s} \xrightarrow{f''} \mathbf{d}{:}\mathit{request}.\mathbf{d}{\downarrow}f''.\mathbf{s}{\uparrow}f''.\mathbf{s}{\downarrow}f'.\mathbf{c}{\downarrow}f_0.\mathsf{end}\end{aligned}$$

The session is started by a call on $\mathbf{c}.\mathit{pubGrd}$*, while other objects are inactive at the moment. After the call* $\mathbf{c} \xrightarrow{f} \mathbf{d}{:}\mathit{publish}$*, the service desk* $\mathbf{d}$ *is active at computing* f *in a process running publish. We position* $\mathbf{d}{\downarrow}f$ *there to specify that* $\mathbf{d}$ *must resolve* f *after it is called by* $\mathbf{c}$ *and before it is called by* $\mathbf{s}$ *(i.e.* $\mathbf{s} \xrightarrow{f''} \mathbf{d}{:}\mathit{request}$*). For* $\mathbf{c}$*, it can have a second remote call* $\mathbf{c} \xrightarrow{f'} \mathbf{s}{:}\mathit{announce}$ *after its first call. As* $\mathbf{d}$ *is called by* $\mathbf{s}$*,* $\mathbf{d}$ *can start computing* f'' *in a process running request only after* $\mathbf{d}{\downarrow}f$*, which means the process computing publish has terminated.* $\mathbf{s}$ *will fetch the result by* $\mathbf{s}{\uparrow}f''$ *after* $\mathbf{d}$ *resolves* f''*, i.e. the students reads its grades. The* end *is there to ensure all processes in the session terminate. The valid use of futures is examined during generating object types from a global type. E.g.: if* $\mathbf{s}{\uparrow}f''$ *is specified before* $\mathbf{d}{\downarrow}f''$*, the projection procedure will return* undefined *since* f'' *can not be read before being resolved.*

Local Types

Local types describe the communication from the point of view of a single endpoint. We differ between *object* types, which describes the communication visible to an object, and *method* types, which describe the communication visible to a single process. The distinction is necessary, as processes have no information about their interleaving. As the order of process invocations is an important part of a protocol, it does not suffice to describe all method types. The order of process invocations is visible at object-level describe how to compose method types. Both local types share the following syntax:

Definition 2.3.2

$$\begin{aligned}\mathbf{L} \ ::= \ & \mathbf{p}!_f m.\mathbf{L} \mid \mathbf{p}?_f m.\mathbf{L} \mid \mathsf{Put}\ f{:}(\mathtt{C}).\mathbf{L} \mid \mathsf{Get}\ f{:}(\mathtt{C}).\mathbf{L} \mid \mathsf{await}(f,f').\mathbf{L}\\ & \mid \mathsf{React}(f).\mathbf{L} \mid \oplus\{\mathbf{L}_j\}_{j\in J} \mid \ \&_f\{\mathbf{L}_j\}_{j\in J} \mid \mathbf{L}^*.\mathbf{L} \mid \mathsf{skip}.\mathbf{L} \mid \mathsf{end}\end{aligned}$$

We use . to denote sequential composition. The type $\mathbf{p}!_f m$ denotes a sending action of the described endpoint via an asynchronous remote call on method m at endpoint $\mathbf{p}$. The type $\mathbf{p}?_f m$ denotes a receiving action of the describes object which starts a new process computing f by executing method m after a call from $\mathbf{p}$. The resolving $\mathsf{Put}\ f{:}(\mathtt{C})$ and fetching $\mathsf{Get}\ f{:}(\mathtt{C})$ have the same intuitive meaning as their global counterparts. The suspension $\mathsf{await}(f,f')$ means that the process computing f suspends its action until future f' is resolved. The reactivation $\mathsf{React}(f)$ means the process continues the execution with f. The choice operator $\oplus$ in $\oplus\{\mathbf{L}_j\}_{j\in J}$ denotes that the currently active process selects a branch to continue. The offer operator $\&_f$ in $\&_f\{\mathbf{L}_j\}_{j\in J}$ denotes that the object offers branches $\{\mathbf{L}_j\}_{j\in J}$ when f is resolved. The type skip denotes no action and we say $\mathbf{L}.\mathsf{skip} \equiv \mathbf{L} \equiv \mathsf{skip}.\mathbf{L}$.

- *A method type* describes the execution of a single process on a particular future f. It has the following attributes: (1) Its first action is $\mathbf{p}?_f m$ for some $\mathbf{p}$, m, f, and (2) if it has a branching type, the final action in every branch is $\mathsf{Put}\ f{:}(\mathtt{C})$ for some $\mathtt{C}, f$, and (3) it contains no further resolving action or receiving action, and (4) it contains no end.

- *An object type* is a type which is not a method type.

- *A condensed type*, denoted by $\hat{\mathbf{L}}$, where $\mathbf{L}$ is an object type, replaces every action, except receiving and reactivation actions, in $\mathbf{L}$ with skip.

Example 2 *Consider object* $\mathbf{d}$ *in the grading system in Section 2.2. Its method type on future* f*, which is used for calling method publish, is* $\mathbf{c}?_f publish.\mathsf{Put}\ f$.

Its object type is $\boldsymbol{L} = \mathbf{c}?_f publish.\mathsf{Put}\ f.\mathbf{s}?_{f''} request.\mathsf{Put}\ f''.\mathsf{end}$*, and its condensed type is*

$$\hat{\boldsymbol{L}} = \mathbf{c}?_f publish.\mathsf{skip}.\mathbf{s}?_{f''} request.\mathsf{skip}.\mathsf{skip} \quad \equiv \quad \mathbf{c}?_f publish.\mathbf{s}?_{f''} request$$

2.4 Verification

To ensure that the scheduler's behavior follows $\mathbf{L}$, we propose a verification mechanism where a scheduler uses a session automaton [10], as a *scheduling policy*, to model the possible sequence of events.

In this model, whenever the object is idle, the object's scheduler inputs the processes which can be (re-)activated according to the session automaton, which is automatically generated by $\mathbf{L}$. If a labelled transition, which corresponds to an event, can fire in the automaton, the object (re)activates this event. If there are several processes which can run such transition, the scheduler randomly selects one of them. This mechanism is a variant of typestate [42].

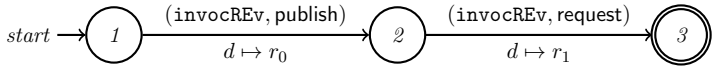
We use *session automata* [10] as our automata model. Session automata work on words over infinite alphabets of the form $\Sigma \times D$, where Σ is finite and D is infinite. In our model D is the set of all futures and $\Sigma = \{\texttt{reactEv}, \texttt{invecREv}\}$ the set of labels, which denote whether a process is started or reactivated. A configuration of a *k-Register Session Automaton* has, additionally to a state, k registers which may store one element from D. Upon transition, the automata may check whether the future of the next letter is equal to any of its registers and store it in one of the registers if it is fresh. Session automata only store fresh futures; a future is only stored if it is the first time it occurs in a word. It is decideable whether two session automata accept the same language [10]. Given an object type, we can build a session automaton:

Definition 2.4.1 *Let* $\boldsymbol{L}$ *be an object type. Let* k *be number of futures in* $\hat{\boldsymbol{L}}$*. We assume the futures are ordered and* $pos(f)$ *refers to the number of* f *in the ordering. The* k*-register session automaton* $A_{\boldsymbol{L}}$ *is defined inductively as follows:*

- $\mathbf{p}?_f m$ *is mapped to a 2-state automaton which reads* $(\texttt{invocREv}, m)$ *and stores the future* f *in the* $pos(f)$*-th register on its sole transition.*
- $\mathsf{React}\ (f)$ *is mapped to a 2-state automaton which reads* `reactEV` *and tests for equality with the* $pos(f)$*-th register on its sole transition.*
- *Concatenation, branching, and repetition using the standard construction for concatenation, union, and repetition for NFAs.*

When a process is activated, the automaton stores the process's corresponding futures; when a process is reactivated, the automaton compares the process's corresponding futures with the specified register. As all repetitions in types projected from a global type are self-contained, after the repetition, the futures used there are resolved and thus the automaton can overwrite it safely. The example below shows how a session automaton works based on an object type:

Example 3 *Consider the example from Section 2.2. A simple automaton describing the sequence for the* $\mathbf{d}$ *(Service Desk) is*



Where $d \mapsto r_i$ *describes that the infinite part of the read letter is copied into register* r_i *The scheduler above does not need to read registers because it does not have reactivations.*

We give a type system that checks ABS systems (including Main Block) against a global type **G**. Methods are checked against the corresponding method type with rather standard typing rules, while on object creations we check whether the annotated scheduler describes the same language as the session automata derived from **G**. Our main theorem can be phrased as:

Theorem 2.4.2 *Let S be an ABS system and $\boldsymbol{G}$ an admissable global type. If h is a sequence and O an object, then $h \upharpoonright O$ is the subsequence of h which contains only the events visible to O. If S can be typed with $\boldsymbol{G}$, then every history h that S may generate is a permutation of a history h' described by $\boldsymbol{G}$ and*

$$\forall O \in \mathsf{Ob}.\ h \upharpoonright O = h' \upharpoonright O$$

2.5 Conclusion

Our work shows that session types, despite being studied for concurrency based on the π-calculus, are flexible enough to be used for more complex concurrency models. For future work, we are interested in transferring other results on π-calculus to ABS, e.g. the composition of local types to global types [19] and implementing our proposed verification algorithm. The proposed scheduler is not implemented in the ABS runtime yet, as it does not complement the existing scheduler for full ABS which uses time for its policy.

The paper "Session-Based Compositional Analysis for Actor-Based Languages Using Futures" is based on a technical report [33], which has two major differences:

- Instead of deriving a session automata, the global type is considered verifiable if all permutations are guaranteed to preserve the indistinguishability from a local view
- Instead of a type system, method types are checked by translating them into a class invariant. These can be verified by KeY-ABS.

Chapter 3

Deadlock Analysis for ABS

3.1 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 paradigmatic 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 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 *behavioural types*, from code. The inference system is *modular* because it (partially) supports separate inference of modules. To meet precision of behavioural types' analysis, over the year we defined and implemented different techniques: (i) an evaluator that computes a fixpoint semantics and (ii) an evaluator using abstract model checking (both described in [29]) and (iii) a more powerful fixpoint technique described in Deliverable D2.1 [14]. This technique was a later improvement of our approach. But the theory has been completely investigated [28] and its implementation is part of the new release of the tool available in the collaboratory.

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 behavioural types* from ABS code. These behavioural types are similar to those ranging from languages for session types [26] to process behavioural types [35] and to calculi of processes as Milner's CCS or pi-calculus [38, 39]. The inference system mostly collects method behaviours and uses constraints to enforce consistencies among behaviors. Then a standard semi-unification 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 behaviours 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 behavioural types of methods in *synchronised* and *unsynchronised behavioural types*, 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 behavioural types feature recursion and resource creation; therefore their underlying models are infinite state and we developed a powerful technique wich is able to decide the presence of deadlock in a infinite state model.

3.2 The DF4ABS Tool

We extended the ABS suite [45] with an implementation of our deadlock analysis framework (at the time of writing the suite has only the fixpoint analyser, the full framework is integrated in the collaboratory. 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 behavioural type inference rules. There are three main modules that comprise DF4ABS:

1. *Behavioural Type 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 behavioural type inference of method calls; (ii) then it visits again all classes of the program to generate the initial environment that maps methods to the corresponding method signatures; and (iii) it finally revisits the AST and, at each node, it applies the behavioural type 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 [30]. When the solver terminates (and no error is found), it produces a substitution that satisfies the input constraints. Applying this substitution to the generated behavioural types produces the abstract class table and the behavioural type 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 behavioural type (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 behavioural type, 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 [44] for connected components of graphs and we stop the algorithm when a circularity is found.

Regarding the computational complexity, the behavioural type inference system runs in polynomial time with respect to the length of the program in most of the cases [30]. The fixpoint analysis is exponential in the number of cog names in a behavioural type class table (because lams may double their size at every iteration). However, this exponential effect actually bites in practice.

3.3 Assessments

We tested DF4ABS on a number of medium-sized programs as well as on the case studies of Envisage. In all the three case studies analysed (Deliverable D4.1 [15]) there were no deadlock presences. In the ATB case study, the tool did not find any of the necessary conditions for a dependency cycle, namely blocking get operations nor pure await cycles. In the case of the ENG case study, the tool was able to run after resolving some minor bugs related to the presence of futures in generic arguments. The FRH was throughly analysed in collaboration with the industrial counterpart, we finally concluded that the tool was able to correctly infer the program behavior in regards to concurrent operations and to successfully verify the deadlock freedom.

3.4 Conclusion

DF4ABS, being modular, may be integrated with other analysis techniques, as we discussed before more than one technique has been implemented. On the other hand the inference module can be modified for reusing the same techniques for other languages. In fact, we have recently applied our deadlock detection algorithm to the analysis of Java Bytecode programs. In order to do this we had to design and implement the inference of the behavioural types from a Java Bytecode program and use these inferred types as input to the core analysis technique. Theoretically we just need to prove a standard subject reduction theorem, which states the soundness of our inference system, leaving the correctness of the algorithm unaffected.

Chapter 4

Contract-based Verification of Asynchronous Method Calls with Cooperative Scheduling

4.1 Invariant-based Verification with Cooperative Scheduling

The ABS language design makes invariant-based verification of ABS concurrent programs [21, 16] modular. Let us briefly recall why:

The invariant-based verification framework for ABS assumes formal specification at the class level, i.e. for each object implemented in a class C we aim to establish its invariant I_{C}. In ABS it is sufficient to show that each method establishes and maintains its class invariant, hence, reasoning is modular in the following way: it is broken down into proof obligations for one method at a time and it is sufficient to establish the local class invariant of a given method, without the need to look at the global system. The soundness of this approach rests on three properties of the ABS language.

First, in ABS at most one process is active on an object at any time. Together with strong encapsulation, this means that, as long as the execution of a method on object o is not suspended, nobody else can modify o's heap. Strong encapsulation means that the heap of an object o can only be modified by itself. Any other object $o' \neq o$, even if it has the same class as o, must use a setter method of o. Hence, the restriction to one active process per object plus strong encapsulation together guarantee that the invariant of an object running a process can only be broken by that process itself, as long as it doesn't suspend execution.

Thus it is sufficient that all methods of a class establish the class invariant whenever they suspend or terminate execution. In ABS programs the suspension points of processes are explicit in the form of await and suspend statements. As argued above, by strong encapsulation and absence of multi-processing, execution between two scheduling points can be viewed as atomic or *sequential*.

Assume now we want to prove that a class C maintains an invariant I_{C}. Clearly, we need to prove that C's initialization block establishes I_{C}. After class initialization, any method m of C might be executed. Hence, for each m we must show that it maintains I_{C}. We can *assume* that the invariant I_{C} holds when m's execution is started, either, because of class initialization or because it has been re-established by the previously executed process. To ensure that I_{C} is re-established at the beginning of any execution, it is sufficient to show that it is re-established whenever a method finishes a sequential piece of code. In ABS this happens exactly in two places: whenever an await or suspend statement is reached and when a method terminates. Hence, we need to *guarantee*, i.e. to prove, that before each await and suspend statement, as well as when a method returns, its class invariant holds. Finally, what happens when a method resumes execution after an await or suspend statement? Well, just like at method invocation time, it can *rely* on the fact that, whichever method has previously suspended execution, it re-established the class invariant.

To summarize, in order to establish a class invariant I_{C}, we need to prove the following facts:

Definition 4.1.1 (Proof Obligations for Invariants in ABS)

1. *The initalization code of* C *establishes* I_{C}.

2. *For each method* m *in* C*, assuming that* I_C *holds at the beginning of its execution and after each* await *and* suspend *statement:*

   (a) I_C *holds before each* await *and* suspend *statement.*

   (b) I_C *holds when* m *terminates.*

Provided that such a proof is done for each class C in an ABS program P, it can be concluded that the invariant $\bigwedge_{C \in P} I_C$ holds for P [21]. Obviously, this is a specific form of rely-guarantee reasoning [32]. It rests on the following properties of the ABS language:

1. Strong encapsulation, i.e. objects can only directly modify their own heap. To modify the heap of another object, a setter function must be used.

2. No multi-processing, i.e. at most one process per object can be active at any time.

3. Explicit scheduling points, i.e. it is syntactically recognizable when a method suspends its execution.

The invariant-based verification framework sketched above was implemented in the deductive verification system KeY-ABS [20, 16], a variant for ABS of the state-of-art verification system KeY [3]. In contrast to model checking, deductive verification, permits to prove invariants for systems with an *unbounded* number of objects, messages, and data size [22, 17]

4.2 Contract-based Verification

The invariant-based verification framework described above and implemented in the ABS tool suite [20, 16] has a serious limitation. It assumes that all methods of a class maintain the same invariant. In other words, the granularity of specifications is situated at the class level. In many applications this is too coarse, because it is necessary to capture the different behaviours embodied by different methods in a single invariant. This results in invariants that are unwieldy, hard to come up with, and unnecessarily difficult to prove.

When verifying sequential programs it is, therefore, common to specify separately the behaviour of each method with the help of a *method contract*. This approach was first explored by Meyer [37] as a general design principle and in the context of runtime monitoring. It is currently the de-facto standard for specification of sequential OO-imperative programs [11]. It is also implemented in the KeY verification system [3] for sequential JAVA programs.

Let us look at the example in Figure 4.1 which features several methods that are specified by a contract. A *contract* consists of three components:

1. A *precondition*, also termed *requires* clause, that specifies the condition under which the called method promises the behaviour specified in the second and third component.

2. A *postcondition*, also termed *ensures* clause, that specifies the condition the called method promises to establish after it terminated.

3. A *modifies* clause, also termed *assignable* clause, that specifies all heap locations whose values the called method might have changed.

We use the notation R_m for the requires clause of a method m and similar for the other specification elements. Each method in Figure 4.1 has a contract. In addition, a class invariant is given. In a sequential setting there is a single processor, and the only points when a method suspends execution are when it calls another method and when it terminates. This leads to a form of rely-guarantee reasoning where we need to prove the following facts:

Definition 4.2.1 (Proof Obligations for Contracts of Sequential Programs)

Example:

```
interface BankI {
  Unit deposit(Int amount);
  Unit withdraw(Int amount);
}

class Bank implements BankI {
  /*@ invariant balance >= 0; *@/
  Int balance = 0;

  /*@ requires amount > 0;
    @ ensures balance == \old(balance) + amount;
    @ assignable balance;
    @*/
  Unit deposit(Int amount) { balance = balance + amount; return; }

  /*@ requires amount > 0 && balance - amount >= 0;
    @ ensures balance == \old(balance) - amount;
    @ assignable balance;
    @*/
  Unit withdraw(Int amount) { if (balance - amount >= 0) {balance = balance - amount;} return;}

  /*@ requires i > 0;
    @ ensures balance > 0;
    @ assignable balance;
    @*/
  Unit process(Int i) {
    this.deposit(2*i);
    this.withdraw(i);
    return;
  }
}
```

Figure 4.1: Specification with contracts

1. *The initialization code of* C *establishes* I_C.
2. *For each method* m *in* C*, assuming that* $I_C \wedge R_m$ *holds at the beginning of its execution and also* $I_C \wedge E_n$ *holds after each call to any method* n *called inside* m*:*
   (a) $I_C \wedge R_n$ *holds before each call to a method* n *inside* m.
   (b) $I_C \wedge E_m$ *holds when* m *terminates.*

Requires clauses of the callee must be proven on the caller side and can be assumed by the callee. Vice versa, ensures clauses are proven on the caller side and the callee's ensures clauses can be assumed by the caller. If we are able to prove this for all classes of a program, then we know that:

1. Each method m respects its contract, i.e. if m is called in state that satisfies R_m, then m terminates and in the final state E_m holds.
2. Each class invariant holds at the beginning and after termination of each of its methods.

For the example, we can easily establish the proof obligations stated above. Let us look at the process() method. At its start we assume its requires clause i > 0 and the class invariant balance >= 0. Assume the current value of the field balance is b. First we prove the requires clause of the callee deposit() (obviously, the class invariant still holds). This follows, because i > 0 implies 2*i > 0. After the call we again assume the class invariant plus the callee's ensures clause: balance == b + 2*i. We also still know that i > 0, because ABS has call-by-value semantics. Now we need to prove the requires clause of the callee withdraw()

(again, the class invariant continues to hold). Considering renaming of formal parameters, we must establish i > 0 && b + 2∗i − i >=0 which, using the class invariant, obviously holds. After the call we again assume the class invariant plus the callee's ensures clause: balance == b + 2∗i −i which clearly implies the ensures clause of process().

So far, the reasoning makes no use of assignable clauses. They can make verification more modular. Assume that the parameter i in the example were declared as a field. Then, after the call to deposit() we loose the information that i > 0, because deposit() could have overwritten i. The assignable clause of deposit() tells us that this is not the case, hence we can still make use of i > 0.

The proof obligations for contract-based verification with assignable clauses are as follows, where the additional items compared to Def. 4.2.1 are highlighted:

Definition 4.2.2 (Proof Obligations for Contracts with Assignable Clauses, Sequential)

1. *The initalization code of* C *establishes* I_C.
2. *For each method* m *in* C*, assuming that* $I_C \wedge R_m$ *holds at the beginning of its execution and also* $I_C \wedge E_n$ *holds after each call to any method* n *called inside* m*:*
   (a) $I_C \wedge R_n$ *holds at each call to a method* n *inside* m.
   (b) $I_C \wedge E_m$ *holds when* m *terminates.*
   (c) ***When*** m ***terminates at most values of heap locations listed in*** A_m ***have changed their value.***

   In addition we may assume that any condition J ***that holds before a call to a method*** n ***inside*** m***, and that does not depend on*** A_n***, still holds after*** n ***returns.***

4.3 Contract-based Verification with Cooperative Scheduling

Assume that we want to prove the example in Figure 4.1 in an ABS setting with asynchronous method calls as in Figure 4.2.

Example:

```
1  /*@ requires i > 0;
2    @ ensures balance > 0;
3    @ assignable balance;
4    @*/
5  Unit process(Int i) {
6    Fut<Unit> f1 = this!deposit(2*i);
7    await f1?;
8    Unit d = f1.get;
9    Fut<Unit> f2 = this!withdraw(i);
10   await f2?;
11   Unit w = f2.get;
12   return;
13 }
```

Figure 4.2: Specification of asynchronous method calls with contracts

The await statements in the code above ensure that deposit() is executed before withdraw() which should be sufficient to establish the same postcondition as above. But it is not obvious how the contracts can be reformulated as invariants. And even if we manage to do so, the resulting specification would be not very readable. Therefore, it seems a natural idea to combine invariant-based verification with asynchronous calls with contract-based verification as known from sequential programming, see the diagram in Figure 4.3.

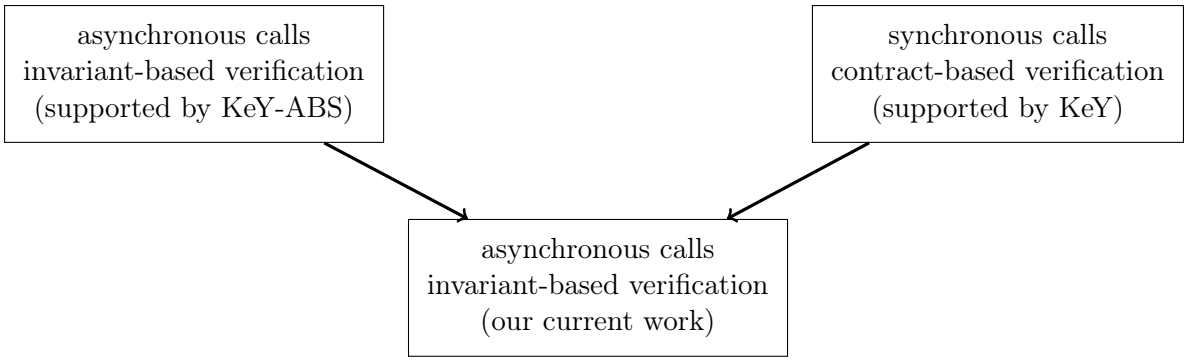


Figure 4.3: Combination of two approaches

Such a combination seems reasonable, because we can already treat ABS code between suspension points as essentially sequential, see Section 4.1. Indeed, Defs 4.1.1 and 4.2.1 have exactly the same structure. We can combine both to obtain proof obligations for programs with asynchronous calls. For the sake of simplicity, we leave out suspend statements, whose treatment is unchanged compared to Def. 4.1.1. We also assume without loss of generality that each await statement relates to exactly one future which resolves a call to a method n. Finally, only asynchronous calls are dealt with: synchronous calls still create the same proof obligations as in Def. 4.2.1.

Definition 4.3.1 (Proof Obligations for Contracts with Asynchronous Calls —Unsound—)

1. *The initalization code of* C *establishes* I_{C}.
2. *For each method* m *in* C*, assuming that* $I_{\mathsf{C}} \wedge R_{\mathsf{m}}$ *holds at the beginning of its execution and also* $I_{\mathsf{C}} \wedge E_{\mathsf{n}}$ *holds after each* await *statement resolving an asynchronous call to method* n *inside* m*:*
   (a) R_{n} *holds before each asynchronous call to a method* n *inside* m.
   (b) $I_{\mathsf{C}} \wedge E_{\mathsf{m}}$ *holds when* m *terminates.*

With this definition, it is possible to prove the contract in Figure 4.2 in the same manner as before. Unfortunately, Def. 4.3.1 is unsound. The problem is illustrated in Figure 4.4: process() makes the asynchronous call to deposit(). According to Def. 4.3.1.(2a) it has to establish the precondition R_d of deposit() *at call time*. But in an asynchronous setting there can be a time gap between the call and the actual execution. During this time, any other active process of the object o, where deposit() executes, for example, withdraw() could change the execution state (the values of heap locations) of o. In this particular example, we happen to know that withdraw() cannot possibly be executed before deposit() starts, but Def. 4.3.2 does not use this information.

To achieve soundness, therefore, it is necessary to add a condition which prevents inadvertent changes to the state of an object between call time and execution time. The same has to be done for the return of deposit() when there is a potential time gap until its result (described by E_d) is used.

Definition 4.3.2 (Proof Obligations for Contracts with Asynchronous Calls)

1. *The initialization code of* C *establishes* I_{C}.
2. *For each method* m *in* C*, assuming that* $I_{\mathsf{C}} \wedge R_{\mathsf{m}}$ *holds at the beginning of its execution and also* $I_{\mathsf{C}} \wedge E_{\mathsf{n}}$ *holds after each* await *statement resolving an asynchronous call to method* n *inside* m, where E_{n} does not depend on the values of fields of the object where it is executed*:*
   (a) R_{n} *holds before each asynchronous call to a method* n *inside* m, where R_{n} does not depend on the values of fields of the object where it is executed.

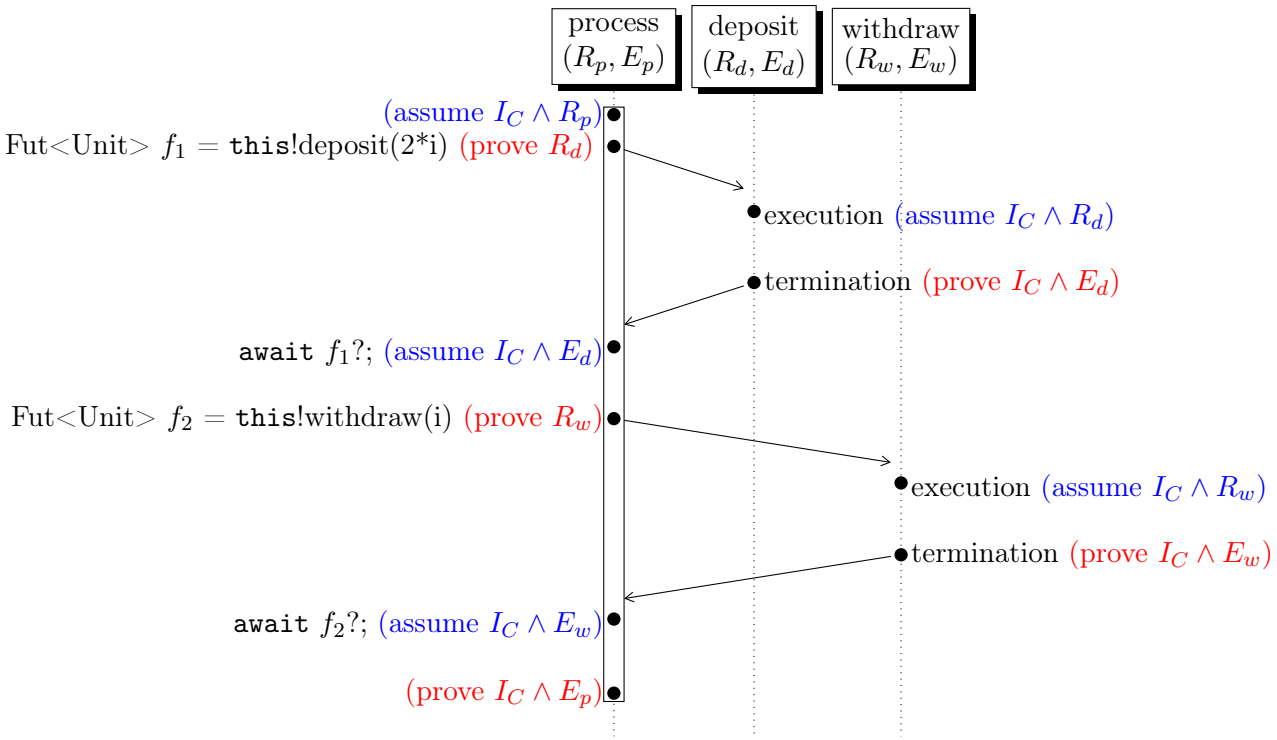


Figure 4.4: Time gap around asynchronous method calls and their execution on an instance of class `Bank`

(b) $I_C \wedge E_m$ *holds when* m *terminates.*

Unfortunately, this sound verification framework allows no longer to prove the example, because the balance field could have been changed after E_d was proven and before it is assumed. As mentioned above, if our example has no other methods than the ones given above, then the await in Line 7 of Figure 4.2 prevents the call to withdraw() to be issued before E_d is used.

At this point, different strategies can be taken. We can liberalize the soundness condition by taking assignable clauses into account:

Definition 4.3.3 (Proof Obligations for Contracts with Asynchronous Calls and Assignable)

1. *The initalization code of* C *establishes* I_C.
2. *For each method* m *in* C*, assuming that* $I_C \wedge R_m$ *holds at the beginning of its execution and also* $I_C \wedge E_n$ *holds after each* await *statement resolving an asynchronous call to method* n *inside* m*, where* E_n *does not depend on the values of fields* mentioned in assignable clauses of any method in the class of m*:*
   (a) R_n *holds before each asynchronous call to a method* n *inside* m*, where* R_n *does not depend on the values of fields* mentioned in assignable clauses of any method in the class of m.
   (b) $I_C \wedge E_m$ *holds when* m *terminates.*

This does not help in the example, because balance occurs in the assignable clause of all methods of class Bank. We need to make use of the information that is implicit in the scheduling points of a program. In ABS we have available a *May-Happen-in-Parallel* (MHP) analysis [6] which provides exactly what is required. Specifically, for each code location l and method m in a class C we can efficiently compute an over-approximation of the set of methods that can possibly be executed in parallel with l, denoted $MHP(l, m)$. In practice, the obtained bounds tend to be rather precise. For example $MHP(7, \text{process}) = \emptyset$. Hence we define for code location l and method m:

$$A(l, m) = \bigcup_{n \in MHP(l,m)} A_n$$

This serves as the basis of an improved definition of proof obligations in a contract-based setting:

Definition 4.3.4 (Proof Obligations for Contracts with Asynchronous Calls and MHP)

1. *The initalization code of* C *establishes* I_{C}.
2. *For each method* m *in* C*, assuming that* $I_{\mathsf{C}} \wedge R_{\mathsf{m}}$ *holds at the beginning of its execution and also* $I_{\mathsf{C}} \wedge E_{\mathsf{n}}$ *holds after each* await *statement at code location* l *resolving an asynchronous call to method* n *inside* m*, where* E_{n} *does not depend on the values of fields in* $A(l, \mathsf{m})$*:*
   (a) R_{n} *holds before each asynchronous call to a method* n *at code location* l *inside* m*, where* R_{n} *does not depend on the values of fields that are potentially modified before method* n *is scheduled.*
   (b) $I_{\mathsf{C}} \wedge E_{\mathsf{m}}$ *holds when* m *terminates.*

Now $A(7, \mathsf{process}) = \emptyset$, therefore E_d is "safe". The same analysis is used to establish R_w at execution time and E_w after resolution.

4.4 Open and Closed World Verification

Let us extend our example by adding a setter method:

Example:

```
Unit setBalance (Int b) { this.balance = b; }
```

As long as setBalance() is never called, or only called outside the context of process, the verification goes through exactly as before. But in general some new code might well affect the set $MHP(l, \mathsf{m})$ at a critical code location, thus stopping a verification argument from being provable. This phenomenon is an instance of the well-known *open* vs. *closed* world assumption in concurrent programming.

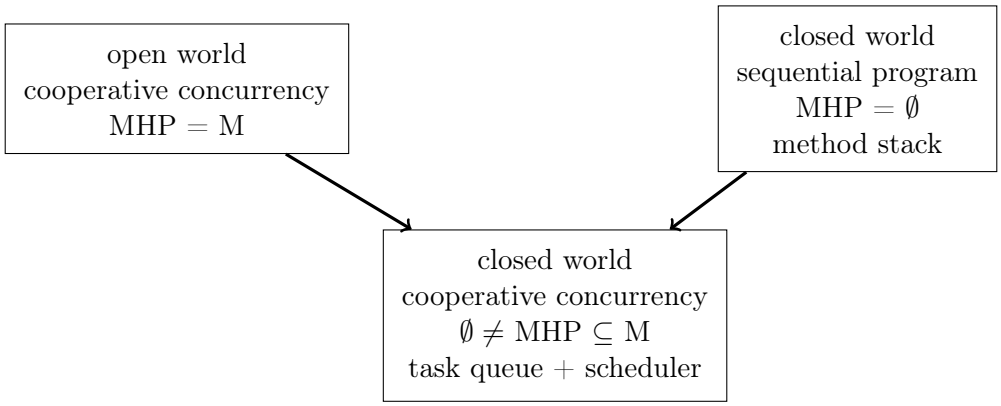


Figure 4.5: Open and closed world analysis

Purely invariant-based verification, as sketched in Sect. 4.1, is suitable for *open world verification*, because it does not need to consider any MHP information. Another way to state it is that invariant-based verification works even if we assume $MHP(l, \mathsf{m}) = M$, where M is the set of all methods in the class of m. On the other hand, in a *sequential* setting, by definition $MHP(l, \mathsf{m}) = \emptyset$: methods are strictly executed in the sequence fixed by the call stack. The situation is illustrated in Figure 4.5.

For contract-based verification we need to move to *closed world*, where the program under analysis is considered to be *fixed*. This allows us to exploit non-trivial analyses, such as MHP, to enlarge the set of verifiable programs.

4.5 A Calculus for Contract-Based Verification of Asynchronous Calls

We develop a contract-based deductive verification system for constructing formal proofs about ABS programs. ABSDL, i.e., first-order dynamic logic for ABS [16, 20], specifies ABS programs together with their intended behaviour in a single logic language. For a sequence of executable ABS statements S and ABSDL formulas P and Q, the formula $P \rightarrow [\![\mathsf{S}]\!]Q$ expresses: If the execution of S starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. In sequent notation $P \rightarrow [\![\mathsf{S}]\!]Q$ is written $P \vdash [\![\mathsf{S}]\!]Q$. The following sequent below shows the proof obligation for the contract of an ABS method m implemented in class C:

$$(I_{\mathsf{C}} \wedge R_{\mathsf{m}}) \vdash [\![methodFrame(source \leftarrow \mathsf{m}, return \leftarrow (var : ret, fut : u)) : S]\!](I_{\mathsf{C}} \wedge E_{\mathsf{m}})$$

The program rules of the sequent calculus realized in ABSDL constitute a symbolic interpreter for ABS. For example, the assignment rule is

$$\mathsf{assign}\ \frac{\vdash \{u\}\{\ell := e\}[\![r]\!]\Psi}{\vdash \{u\}[\![\ell = e;\ r]\!]\Psi}$$

where ℓ 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* [3]. Updates are expressions of the form $\{\ell := e\}$ which captures state changes. Symbolic execution continues with the remaining program r. Updates can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. Updates can only be applied to formulas or terms. Once the program to be verified has been completely symbolically executed and the modality is empty, the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. The rule for the conditional statement is

$$\mathsf{ifElse}\ \frac{\vdash \{u\}(b \rightarrow [\![s_1;r]\!]\Psi) \qquad \vdash \{u\}(\neg b \rightarrow [\![s_2;r]\!]\Psi)}{\vdash \{u\}[\![\mathsf{if}\ (b)\ \mathsf{then}\ \{s_1\}\ \mathsf{else}\ \{s_2\};\ r]\!]\Psi}$$

where the proof tree is split into two cases, one for each possible value of the boolean guard b.

We discuss now the rules needed to support contract-based verification. We start with the symbolic execution of an asynchronous method invocation on the current object:

$$\mathsf{async}\ \frac{\begin{array}{l}\vdash \{u\}\{\overline{x} := \overline{e}\}R_{\mathsf{m}} \\ \vdash \{u\}\{h := h \circ invocEv(\mathtt{this}, \mathtt{this}, f', m, \overline{e})\}\{q := q \circ f'\}\{f := f'\}(isFresh(f') \rightarrow [\![r]\!]\Psi)\end{array}}{\vdash \{u\}[\![f = \mathtt{this}!m(\overline{e});\ r]\!]\Psi}$$

The first premiss proves the precondition of method m. The declaration of method m is "$T\ \mathsf{m}(\overline{T}\ \overline{x})$". The second premiss introduces a constant f' which represents the future generated for the result of this method invocation. This premiss verifies the remaining program based on the assumption that f' is fresh. The execution of this statement extends the current execution history h with an invocation event [21] and appends future f' to the sequence of active futures q.

Before we discuss the next rule we introduce the helper function *mod*, which over-approximates the set of all fields that are possibly modified by tasks executed in parallel to a task running at a given program location. The function $mod : PrgLocation \times PrgElement \times Future \rightarrow Set\langle Field\rangle$ is defined as

$$\begin{array}{l} mod(lc, \mathtt{skip}, z) = \emptyset \\ mod(lc, v = exp, z) = \emptyset \text{ where } v \text{ is a local variable} \\ mod(lc, \mathtt{this}.a = exp, z) = \{a\} \\ mod(lc, (lc_p : p);(lc_q : q), z) = mod(lc_p, p, z) \cup mod(lc_q, q, z) \\ mod(lc, \mathtt{if}\ (b)\ \mathtt{then}\ (lc_{p_1} : p_1)\ \mathtt{else}\ (lc_{p_2} : p_2), z) = mod(lc_{p_1}, p_1, z) \cup mod(lc_{p_2}, p_2, z) \\ mod(lc, \mathtt{while}\ (b)\ \{lc_{body} : body\}, z) = mod(lc_{body}, body, z) \\ mod(lc, \mathtt{await}\ f?, z) = \bigcup_{\mathsf{n} \in project(MHP(lc, method(lc)), \mathtt{this}) \setminus sub(lc, f, z)} A_{\mathsf{n}} \end{array}$$

The first two lines are the base cases for skips and local variable assignments. Neither modifies the heap, hence, the empty set of fields is returned. In case of a field assignment the modified field a is added. Except for the last case the other lines are just the homomorphic extension of the function to composed statements (in ABS the heap is object-local, so it is sufficient to look at the loop body once without the need for a fixed-point computation).

The last case applies MHP analysis to estimate all the possible methods that may run in parallel with the await statement. Function $project(s, o)$ projects the set of methods s to object o. Function $mtd(f)$ returns the method of the future f. We need to take the method to which the future f belongs into account, but the current code is part of a method m executed by a future different from f. This method cannot be parallel to itself and is subtracted only if the analysis yields that the method may not be executed at the await by a task which resolves a different future. Formally, we define the function

$$sub(lc, f, z) = \{\mathsf{m} \mid \mathsf{m} = mtd(z) \wedge z = f \wedge \ \mathsf{m} \text{ not executed by another task while waiting at } lc\}$$

The rule for an await statement on an *unresolved* future is:

$$\text{awaitRelease}\ \frac{\begin{array}{l}\vdash \{u\}(f \in q)\\ \vdash \{u\}I_{\mathsf{C}}\\ \vdash \forall z \in \{u\}q\ .\ \big(Dep(R_{mtd(z)}) \cap \bigcup_{\widehat{lc_z} < x \leq lc} mod(x, prgAt(x), z)\big) = \emptyset\\ \vdash \{u\}\{u_a\}(I_{\mathsf{C}} \wedge \{\widehat{u_a}\}\{u_{arg}\}E_{\mathsf{m}} \rightarrow [\![r]\!]\Psi)\end{array}}{\vdash \{u\}[\![lc : \mathsf{await}\ f?;\ r]\!]\Psi}$$

where

- $Dep(\phi)$ returns the set of all heap locations (fields) that occur syntactically in formula ϕ.
- m is the method which will resolve future f
- The notation $l : q$ denotes that the location of program element q is l, for instance, the await statement in the conclusion is located at lc. The location $\widehat{lc_z}$ is the closest location before lc such that the statement at $\widehat{lc_z}$ is either a release point or the asynchronous call creating future z. Not every await statement is a release statement (in case the await is on an already resolved future). To distinguish these cases our calculus maintains an implicit list $awaitS$ of awaits it knows to be no release points. This list is updated by rule awaitSkip and used here to distinguish "real" awaits from disguised skips.
- The update u_{arg} is defined as $arg_1 := getArgument(hist, f, 1) \| \ldots \| arg_j := getArgument(hist, f, j)$, where m is declared as $T\ \mathsf{m}(T_1\, arg_1, \ldots, T_j\ arg_j)$ and function $getArgument$ extracts the values of the arguments at invocation time of the method resolving future f from the history.
- u_a is the anonymizing update assigning all heap locations possibly changed during the await statement a fresh value, i.e., all locations in the set $mod(lc, \texttt{await}f?, f) \cup A_m$. The update $\widehat{u_a}$ is a similar with other fresh constants, but merely for locations in $mod(lc, \texttt{await}f?, f)$.

The first premiss shows that future f is in the active future set. The second premiss shows that the class invariant should hold before the current process is released. The third premiss proves that none of the preconditions of active futures is violated by any of the method executions between $\widehat{lc_z}$ and now and will not be violated by any possible parallel execution at this await statement. The dependency function $Dep(R_{\mathsf{n}})$ simply returns the set of fields contained in the precondition R_{n}. The final premiss

$$\vdash \{u\}\{u_a\}(I_{\mathsf{C}} \wedge \{\widehat{u_a}\}\{u_{arg}\}E_{\mathsf{m}} \rightarrow [\![r]\!]\Psi)$$

requires more explanation. It is concerned with the situation when we resume control after the await statement. The execution state at resume time is described by the updates $\{u\}\{u_a\}$, where u_a anonymizes all fields (of this object) which might have been changed after release and before resuming control. At any

release point the class invariant must be re-established (second premiss), so we know that I_{C} holds in the state $\{u\}\{u_a\}$.

In addition we know that the task executing method m to resolve future f finished at some point between release and resume. At that point the postcondition E_{m} of m was satisfied. But from this we cannot simply deduce that $\{u\}\{u_a\}\{u_{arg}\}E_{\mathsf{m}}$ holds (u_{arg} evaluates the method arguments mentioned in E_{m} to their correct values).

The reason is that after execution of m finished other tasks might have been scheduled that changed the value of fields occurring in E_{m}. These changes are reflected in $\{u\}\{u_a\}$. Consequently, we have to anonymize all fields in E_{m} again. This is achieved by update $\widehat{u_a}$, but why do we use a different update than u_a?

In many cases $\widehat{u_a}$ will anonymize the same fields as u_a. But the definition of $\widehat{u_a}$ does not add the locations from A_{m}, hence, if some of these locations are not in the set returned by $mod(lc, \mathtt{await} f?, f)$, they will not be anonymized. Intuitively, this is because those fields may only be modified by the task resolving f and not by any other currently active task. Their values are identical at the point when execution of m is finished and when the execution is resumed. We can regain some precision here by using $\widehat{u_a}$.

For example, assume that at the await statement, we have two tasks running, one resolving future f by executing method m and one resolving a different future g by executing method n. Let $A_{\mathsf{m}} = \{a, b\}$ be the assignable clause for m and $A_{\mathsf{n}} = \{a\}$ be the assignable clause for n. In that case $mod(lc, \mathtt{await} f?, f)$ returns the set $\{a\}$. The update u_a is $\{a := c \parallel b := d\}$, anonymizing the location in the union of the sets $mod(lc, \mathtt{await} f?, f) \cup A_{\mathsf{m}} = \{a, b\}$ as both locations might have been changed upon resume.

We know the ensures clause of m holds at some point in time, but we do not know in which order m and n were scheduled (or whether n was scheduled at all). But in any case n changes at most location a and not b and that knowledge is reflected in the update $\widehat{u_a}$ which anonymizes only the location returned by $mod(lc, \mathtt{await} f?, f) = \{a\}$. Hence, it only overrides the value of a when evaluating E_{m} but not of b.

The rule for an await statement on a *resolved* future is:

$$\mathsf{awaitSkip}\ \frac{\vdash \{u\}(f \notin q) \qquad \vdash \{u\}[\![r]\!]\Psi}{\vdash \{u\}[\![lc : \mathsf{await}\ f?;\ r]\!]\Psi} \qquad awaitS := awaitS \circ lc$$

This await statement is located at lc. The rule also records that this await statement is equivalent to a skip statement: $awaitS := awaitS \circ lc$. As explained above, this information is required to calculate $\widehat{lc_z}$ in rule awaitRelease. The rule for the return statement is:

$$\mathsf{return}\ \frac{\vdash \{u\}\{h := h \circ compEv(\mathtt{this}, f, \mathsf{m}, e)\}\{q := q \setminus f\}\Psi}{\vdash \{u\}[\![\mathsf{return}\ e]\!]\Psi}$$

This rule removes the modality box, extends the history with a completion event [21], and subtracts the current future from the active future set. The postcondition Ψ should hold at this final state captured by the updates.

Discussion of the Banking Example Consider the process() method listed in Figure 4.2. The precondition of method deposit() does not contain fields, and it is the only method which can be executed at the await f_1? statement in process(). So the precondition of deposit() will not be violated when it is scheduled for execution. The precondition of withdraw() contains field balance. Since the withdraw() method is the only one that can be executed at the await f_2? statement in process(), its precondition will not be violated when it is scheduled for execution. The process() method listed in Figure 4.2 can, therefore, be proven by our verification framework.

Now we change process() such that the synchronization of futures f_1 and f_2, i.e. awaitf_1?; awaitf_2?, occurs after the asynchronous call to withdraw and we add "balance $\geq i$" to the requires clause of the process() method, see Figure 4.6.

In this case our calculus is too strict to prove the specification of process(). The reason is that we require when withdraw() is scheduled that field balance was not modified after withdraw() was invoked or after the most recent scheduling point. But deposit may be scheduled before withdraw and modify balance.

Example:

```
/*@ requires i > 0 && balance ≥ i;
  @ ensures balance > 0;
  @ assignable balance;
  @*/
Unit process(Int i) {
  Fut<Unit> f1 = this!deposit(2*i);
  Fut<Unit> f2 = this!withdraw(i);
  await f1?;
  await f2?;
  Unit d = f1.get;
  Unit w = f2.get;
  return;
}
```

Figure 4.6: Unprovable variant of the example.

4.6 Related and Future Work

4.6.1 Related Work

The work of [25] implemented a rely-guarantee proof system [1, 32] on top of the Frama-C framework and demonstrated modular proofs of partial correctness on asynchronous programs written in C using the Libevent library. However, it only works for restricted C programs that use this specific library.

4.6.2 Tracking of Active Futures

It turns out that MHP is not precise enough to enable verifiability of some classes of programs. We illustrate this with the tail recursive example in Figure 4.7. Method m calls recursively itself until the field c is equal to n. The execution order between the processes is fixed, because there is no statement after the await, and field c can only be modified by the recursive calls. But an MHP analysis is not sufficient, because it cannot resolve the *order* in which the futures are resolved. Specifically, the rules in the previous section do not distinguish between active futures that wait on the current future (and, therefore, cannot be executed in parallel to it), the active futures that have been resolved (and also cannot be executed in parallel), and the remaining active futures that could actually interfere at a given code location.

To make the calculus in Section 4.5 work for the code in Figure 4.7, we need to do the following:

1. Introduce functions *Blocked* and *Resolved* that for each code location approximate the active futures whose execution is blocked, because it depends on the currently executed future and the active futures that have been already resolved, respectively.

2. Enhance the calculus rules so that the *Blocked* and *Resolved* functions are updated.

3. Enhance the method contracts with requirements and guarantees about blocked and resolved futures. For example method m in Figure 4.7 should ensure that all the futures it creates have been resolved when it terminates.

4.6.3 Block Contracts

Block contracts [3, Chapter 3] can be used to refine the analysis even further: the idea is to introduce a contract for each "atomic" piece of ABS code, i.e. the code between two synchronisation points. This contract then specifies the information that is preserved in that block. In many cases, block contracts might be approximated automatically to a certain extent. For example, the *mod* function in Section 4.5 computes a specific form of a block contract.

Example:

```
1  interface CI { m(); }
2
3  /*@ invariant 0 <= c && c <= n; @*/
4  class C(Int n) implements CI {
5    Int c = 0;
6
7    /*@ requires true;
8      @ ensures c == n;
9      @ assignable c;
10     @*/
11   Unit m() {
12     if (c < n) {
13       Fut<Unit> f = this!m();
14       c = c + 1;
15       await f?;
16     }
17     return;
18   }
19 }
```

Figure 4.7: A tail recursive example

4.6.4 Assumptions on the Scheduler

To address situations such as the one stated at the end of Section 4.5, we plan to investigate how far the constraints on the assignable clauses can be relaxed without sacrificing soundness. We also consider to state assumptions about the scheduler. For example, the scheduler at runtime may only select methods for execution whose preconditions can be satisfied. This opens an interesting direction of research that explores the trade-offs between provability and deadlocks. ABS is ideally suited as a target language, because powerful deductive verification as well as deadlock detection methods are available.

Chapter 5

Combining Static Analysis and Testing for Deadlock Detection

5.1 Introduction

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks, months or years after releasing the application. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives.

In contrast, testing consists in executing the application for concrete input values. Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings. This is known as *systematic testing* [12, 41] in the context of concurrent programs. The primary advantage of systematic testing for deadlock detection is that it can provide the deadlock trace with all information that the user needs in order to fix the problem. There are two shortcomings though: (1) Although recent research tries to avoid redundant exploration as much as possible [2, 5, 12, 23], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming; (2) Since not all inputs can be tried, there is no guarantee of deadlock freedom.

In this line of work, we propose a seamless combination of static analysis and testing for effective deadlock detection as follows: the deadlock analysis of SACO [24] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide the SYCO testing tool (described in Deliverable D3.5) in order to find associated deadlock traces (or discard them).

5.2 Motivating Example

Our running example is a simple version of the classical sleeping barber problem where a barber sleeps until a client arrives and takes a chair, and the client wakes up the barber to get a haircut. Our implementation in ABS of Figure 5.1 has a main method shown on the left and three classes Ba, Ch and Cl implementing the barber, chair and client, respectively. Method main creates three objects barber, client and chair and spawns two asynchronous tasks to start the wakeup task in the client and sleeps in the barber, both tasks can run in parallel. The execution of sleeps spawns an asynchronous task on the chair to represent the fact that the client takes the chair, and then blocks at line 12 (L12 for short) until the chair is taken. The task taken first adds the task sits on the client, and then awaits on its termination at L20 without blocking, so that another task on the object chair can execute. On the other hand, the execution of wakeup in the client spawns an asynchronous task cuts on the barber and one on the chair, isClean, to check if the chair is clean. The execution of the client blocks until cuts has finished.

Fig. 5.1 summarizes the systematic testing tree of the main method. Edges correspond to execution

```
1  main() {
2    Ba barber = new Ba();
3    Cl client = new Cl();
4    Ch chair = new Ch();
5    client!wakeup(barber,chair);
6    barber!sleeps(client,chair);
7  }
8  class Ba{
9    Unit sleeps(Cl cl, Ch ch){
10     Fut<Unit> f;
11     f = ch!taken(cl);
12     f.get;
13     return Unit; }
14   Unit cuts(){
15     return Unit; }
16 }
```

```
17 class Ch{
18   Unit taken(Cl cl){
19     Fut<Unit> f = cl!sits();
20     await f?;
21     return Unit;}
22   Unit isClean(){
23     return Unit;}
24 }
25 class Cl{
26   Unit wakeup(Ba b, Ch ch){
27     Fut f = b!cuts();
28     ch!isClean();
29     f.get;
30     return Unit; }
31   Unit sits(){
32     return Unit;}
33 }
```

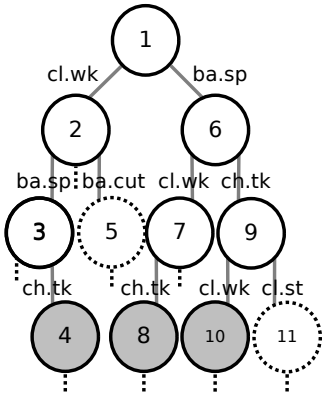


Figure 5.1: Classical Sleeping Barber Problem (left) and Execution Tree (right)

(macro-)steps[1]. Derivations that contain a gray node are deadlocks whereas derivations with a dotted node are complete (non-deadlock) derivations. The main goal of this work is to detect as early as possible that derivations with a dotted node will not lead to deadlock, pruning them as early as possible. Let us observe two selected derivations in detail. In the derivation ending at node 5, the first macro-step executes cl.wakeup and then ba.cuts. At this point, it is clear that object cl will not deadlock, since the get at L29 will succeed and the other two objects will be also able to complete their tasks, namely the await at L20 of object ch can finish because the client is certainly not blocked, and also the get at L12 will succeed because the task in taken will eventually finish as its object is not blocked. However, in the branch of node 4, we first select wakeup (and block client), then we select sleeps (and block barber), and then select taken that will remain in the await at L20 and will never succeed since it is awaiting for the termination of a task of a blocked object. Thus, we have a deadlock.

5.3 Enhanced Semantics with Interleavings Table

We need to extend the standard ABS semantics with information about the task interleavings and the paths taken. This allows us to easily detect *dependencies* among tasks, i.e., when a task is awaiting for the termination of another one. These dependencies are necessary to detect in a second step *deadlock states*, in which there are several concurrent objects involved in a deadlock. Then, we define the *interleavings table* which is a mapping with entries of the form $t_{o,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{o,tk,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the object o and task tk that have been selected in the macro-step, and the program point pp of the first instruction that will be executed;
- n is an integer representing the time when the macro-step starts executing;
- ρ is the status of the task after the macro-step and it can take three values: get or await when executing these instructions on a future variable that is not ready; **return** that allows us to know that the task finished.

[1] A macro-step execution corresponds to the sequencial execution of the instructions in the selected task of the selected object until the next release point (i.e., an await/get instruction or the return).

Example 4 *The interleavings table below (left) is computed for the derivation related to the branch of node 4 in Figure 5.1. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The right column shows the future variables in the state that store the object and task they are bound to.*

St_1	$t_{ini,main,1} \mapsto \langle 0, return \rangle$	$\emptyset$
St_2	$t_{cl,wakeup,26} \mapsto \langle 1, 29{:}f_0.get \rangle$	$fut(f_0, ba, cuts, 14)$
St_3	$t_{ba,sleeps,9} \mapsto \langle 2, 12{:}f_1.get \rangle$	$fut(f_1, ch, taken, 18)$
St_4	$t_{ch,taken,18} \mapsto \langle 3, 20{:}await\ f_2? \rangle$	$fut(f_2, cl, sits, 31)$

The role of this *interleavings table* is twofold: (1) It stores a time identifier and all decisions about task interleavings made during such time. This way, at the end of a concrete execution, the exact ordering of the performed macro-steps can be observed. (2) It will be used to detect deadlocks as early as possible, and, also to detect states from which a deadlock cannot occur, therefore allowing to prune the execution tree when we are looking for deadlocks.

Now, we present the notion of a *deadlock state* which characterizes states that contain a call chain in which one or more tasks are waiting for each other's termination and none of them can make any progress. Thanks to the information store by the *interleavings table*, we can easily detect these states.

Example 5 *Following Ex. 4, St_4 is a deadlock state since there exists a* call chain $\{t_{cl,wakeup,26}, t_{ba,sleeps,9}, t_{ch,taken,18}\}$. *For the first element $t_{cl,wakeup,26}$, the task* wakeup *is blocking the object* cl *and we have task* cuts *waiting on an answer that* b *must provide. At the same time, for the second element of the chain, $t_{ba,sleeps,9}$, the task* sleeps *is waiting that the object* ch *is taken and, finally, the task* taken *is blocking* ch *until* cl *sits. But this one can never sit, as it is blocked and, thus, we got a deadlock. .*

Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit. Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using this naive extension. The early detection of deadlocks is crucial to reduce state exploration as our experiments show in Section 5.5.

5.4 Combining Static Deadlock Analysis and Testing

We propose a deadlock detection methodology that combines static analysis and systematic testing as follows. First, the result of a deadlock analysis of SACO of [24] is a run, which provides a set of abstractions of potential *deadlock cycles*. The nodes of these cycles are abstract objects and tasks and the edges, the different relations that could arise among them. They are abstractions since there could be many objects created at the same program point and all of them are represented by the same abstract object and many tasks executing the same method represented by the same abstract task. If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we systematically test the program using a novel technique to guide the exploration towards paths that might lead to deadlock cycles. The goals of this process are: (1) finding concrete deadlock traces associated to the feasible cycles, and, (2) discarding unfeasible deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input or, in our case, for the main method under test. As our experiments show in Section 5.5, our technique allows reducing significantly the search space compared to the full systematic exploration.

Example 6 *In our working example there are three abstract objects, o_2, o_3 and o_4, corresponding to objects* barber, client *and* chair, *created at lines* 2, 3 *and* 4*; and six abstract tasks, sleeps, cuts, wakeup, sits, taken and isClean. The following cycle is inferred by the deadlock analysis: $o_2 \xrightarrow{12:sleeps} taken \xrightarrow{20:taken} sits \xrightarrow{31:sits} o_3 \xrightarrow{29:wakeup} cuts \xrightarrow{14:cuts} o_2$. The first arrow captures that the object created at L2 is blocked waiting for the termination of task* taken *because of the synchronization at L12 of task* sleeps. *Observe that cycles contain dependencies also between tasks, like the second arrow, where we capture that* taken *is waiting for* sits. *Also,*

a dependency between a task (e.g., sits) and a object (e.g., o_3) captures that the task is trying to execute on that (possibly) blocked object. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can observed, it is complex to figure out from them why these dependencies arise, and in particular the interleavings scheduled to lead to this situation.

Given an abstract deadlock cycle, we guide the systematic execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied.

Example 7 *The following deadlock-cycle constraints are computed for the cycle in Ex. 6:* $\{ \exists t_{O_1,Tk_1,\_} \mapsto \langle \_, 12{:}F_1.get\rangle,\ \exists\ fut(F_1, O_2, Tk_2, 18),\ \exists t_{O_2,Tk_2,\_} \mapsto \langle \_, 20{:}await\ F_2?\rangle, \exists\ fut(F_2, O_3, Tk_3, 31),$ $\mathsf{pending}(Tk_3),\ \exists t_{O_3,Tk_4,\_} \mapsto \langle \_, 29{:}F_3.get\rangle, \exists fut(F_3, O_4, Tk_5, 14), \mathsf{pending}(Tk_5)\}$. *The first four constraints require the* interleavings table *to contain a concrete time in which* some *barber sleeps waiting at L*12 *for a* certain *chair to be taken at L*18 *and, during another concrete time, this one waits at L*20 *for a* certain *client to sit at L*31. *The client is not allowed to sit by the constraint* $\mathsf{pending}(Tk_3)$. *Furthermore, the last three constraints require a concrete time in which* this *client waits at L*29 *to get a haircut by* some *barber at L*14 *and that haircut is never performed, because of the constraint* $\mathsf{pending}(Tk_5)$.

Finally, our semantics is extended to check if at least a *deadlock cycle constraint* does not hold. Intuitively, a *deadlock cycle constraint* does not hold if: (i) the program point which generates the contraint is not reachable anymore, and, (ii) in the case that the interleavings table contains entries matching it, for each one, there is an associated future variable in the state and a pending constraint for its associated task which is violated, i.e., the associated task has finished. The first condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table. The second condition (ii) implies that, for each potential cycle in the state, there is no call chain leading to deadlock since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given cycle, hence the derivation can be stopped.

Example 8 *Let us consider the deadlock-guided testing of our working example with the deadlock-cycle of Ex. 6, and hence with the constraints of Ex. 7. The interleavings table at* St_5 *contains the entries* $\{t_{ini,main,1} \mapsto \langle 0, return\rangle,\ t_{cl,wakeup,26} \mapsto \langle 1, 29{:}f_0.get\rangle$ *and* $t_{ba,cuts,14} \mapsto \langle 2, return\rangle\}$. *One of the constraints does not hold since* $t_{O_1,Tk_1,29}$ *is not reachable (L25 is not reachable anymore) from* St_5 *and constraint* $\mathsf{pending}(Tk_5)$ *is violated (task cuts of the last entry has already finished at this point). The derivation is hence pruned. Similarly, the rightmost derivation is stopped at* St_{11}. *Also, derivations at* St_4, St_8 *and* St_{10} *are stopped by the semantics, as all of them are* deadlock states. *Our methodology therefore explores* 19 *states instead of the* 181 *explored by the full systematic execution.*

5.5 Experimental Evaluation

We have implemented our approach within the SYCO tool which is integrated in the ABS collaboratory (see Deliverable D3.5). This section summarizes our experimental results which aim at demonstrating the effectiveness and impact of the proposed techniques. The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *SB* is an extension of the sleeping barber, *UL* is a loop that creates asynchronous tasks and concurrent objects, *PA* is the pairing problem, *FA* is a distributed factorial, *WM* is the water molecule making problem, *HB* the hungry birds problem; and, (ii) deadlock free versions of some of the above, named *fX* for the *X* problem, for which deadlock analyzers give false positives. We also include here a peer-to-peer system *P2P*.

	Systematic			DGT (deadlock-per-cycle)					Speedup			
Bm.	**Ans**	T	S	**D/U/C**	T	T_{max}	S	S_{max}	T_{gain}	S_{gain}	T^{max}_{gain}	S^{max}_{gain}
HB	35k	32k	114k	2/3/5	44k	15k	103k	34k	0.73	0.9	2.15	3.33
FA	11k	11k	41k	2/1/3	2k	759	3k	2k	5.5	13.7	15.1	22.2
UL	>90k	>150k	>489k	1/0/1	133	133	5	5	>1.1k	>2.5k	>2.5k	>98k
SB	>103k	>150k	>584k	1/0/1	59	59	23	23	>2.5k	>25k	>2.5k	>25k
PA	>121k	>150k	>329k	2/0/2	42	4	12	6	>3.6k	>27k	>38k	>55k
WM	>82k	>150k	>380k	1/0/2	>150k	>150k	>258k	>258k	1^*	1.47^*	1^*	1.47^*
fFA	5k	7k	25k	0/1/1	5k	5k	11k	11k	1.61	2.35	1.61	2.35
fP2P	25k	66k	118k	0/1/1	34k	34k	52k	52k	1.96	2.28	1.96	2.28
fPA	7k	7k	30k	0/2/2	4k	2k	9k	4k	1.75	3.33	3.73	6.98
fUL	>102k	>150k	>527k	0/1/1	410	410	236	236	>1k	>2k	>1k	>2k

Table 5.1: Experimental results: Deadlock-guided testing vs. systematic testing

Table 5.1 shows, for each benchmark, the results of our deadlock-guided testing (DGT) methodology for finding a representative trace for each deadlock compared to those of the standard systematic testing. Partial-order reduction techniques are not applied since they are orthogonal. This way so we focus on the reductions obtained due to our technique per se. For the systematic testing setting we measure: the number of solutions or complete derivations (column *Ans*), the total time taken (column T) and the number of states generated (column S). For the DGT setting, besides the time and number of states (columns T and S), we measure the "number of deadlock executions"/"number of unfeasible cycles"/"number of abstract cycles inferred by the deadlock analysis" (column $D/U/C$), and, since the DCGTs for each cycle are independent and can be performed in parallel, we show the maximum time and maximum number of states measured among the different DCGTs (columns T_{max} and S_{max}). For instance, in the DGT for *HB* the analysis has found five abstract cycles, we only found a deadlock execution for two of them (therefore 3 of them were unfeasible), $44s$ being the total time of the process, and $15s$ the time of the longest DCGT (including the time of the deadlock analysis) and hence the total time assuming an ideal parallel setting with 5 processors. Columns in the group **Speedup** show the gains of DGT over systematic testing both assuming a sequential setting, hence considering values T and S of DGT (column T_{gain} for time and S_{gain} for number of states), and an ideal parallel setting, therefore considering T_{max} and S_{max} (columns T^{max}_{gain} and S^{max}_{gain}). The gains are computed as X/Y, X being the measure of systematic testing and Y that of DGT. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. A timeout of 150s is used. When the timeout is reached, we write $>X$ to indicate that for the corresponding measure we have got X units in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X. Also, we write X^* when DGT times out.

Our experiments support our claim that testing complements static analysis in the context of deadlock detection. In the case of programs with deadlock, we have been able to provide concrete traces for feasible deadlock cycles and to discard unfeasible cycles. For deadlock-free programs, we have been able to discard all potential cycles and therefore prove deadlock freedom. More importantly, the experiments demonstrate that our DGT methodology achieves a notable reduction of the search space over systematic testing in most cases. Except for benchmarks HB and WM which are explained below, the gains of DGT both in time and number of states are enormous (more than three orders of magnitude in many cases). E.g., by just exploring 23 states in SB we found one representative per cycle in just 59ms, whereas systematic testing times out. It can be observed that the gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in SB, UL and PA). In general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively, and therefore can degrade the overall process as it happens with benchmarks HB and WM. Even in these cases, DGT outperforms systematic testing in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, each DCGT cannot stop until all potential deadlock paths have been considered. As expected, when we consider a parallel setting, the gains are in general much larger.

All in all, we argue that our experiments show that our methodology complements static analysis in the context of deadlock detection, finding deadlock traces for the potential deadlock cycles and discarding unfeasible ones, with a significant reduction of the state exploration compared to systematic testing.

5.6 Conclusions

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [13, 24, 27] and thread-based programs [36, 40], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for handling iteration and pointer aliasing) which might lead to a "don't know" answer. Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove the absence of deadlock. Using our method, if there might be a deadlock, we try to find it by exploring the paths given by our deadlock detection algorithm that relies on the static information.

Finally, although we have presented our technique in the context of dynamic testing, our approach would be applicable also in static testing where the execution is performed on constraints variables rather than on concrete values. This remains as subject for future research.

Chapter 6

Combination of Resource Analysis and Profiling

Resource Analysis and Simulation

As explained in deliverable D1.2.2, the static analyses integrated in the resource analyzer SACO [4] infer upper bounds on the resource consumption of the different methods, using different cost models such as number of executed instructions, size of transmitted data, number of objects created, number of methods invoked, etc. These upper bounds are usually complex (mathematical) cost expressions that involve the size of the method parameters. Consider the following iterative method sum that takes a list of integer numbers and sums all its elements:

Example:

```
1 Int sum(List<Int> list) {
2   Int result = 0;
3   while (list != Nil) {
4     Int h = head(list);
5     result = result + h;
6     list = tail(list);
7   }
8   return result;
9 }
```

Analyzing the above code using SACO, we obtain the following upper bound on the number of executed instructions:

```
10 UB for sum(list) = 6+11*nat(list)
```

As expected, the upper bound of method sum depends only on the parameter list, which represents the length of the list, since it determines the iterations of the loop. Note that nat(v) is interpreted as max (0,v). Although computing upper bounds for concrete data is simple in the specific case above, for more complex data-structures with nesting it is more complicated as the definition of the corresponding sizes is more elaborated: number of nodes in a tree, maximum key size in a map, etc. In order to facilitate the usage of SACO results during simulation, SACO generates ABS functions that compute the corresponding upper bounds of a method from concrete data. These functions take the same parameters as the corresponding methods and return the evaluation of the upper bound with respect to those parameters. For the pow method, SACO would generate the following code:

```
11 def Int nat(Int n) = max(0, n) ;
12
13 def Int ub_sum(List<Int> list) = 6+11*nat(size_list(list)-1);
14
```

```
15 def Int size_list(List<Int> x) =
16   case x {
17     Nil => 1;
18     Cons(x1,x2) => 1 + size_list(x2);
19   };
```

Note that in addition to the functions that define the upper bound of method sum, called ub_sum, SACO generates a function size_list that computes the size of a list of type List<Int> (in this case its length plus 1, as it counts the Nil constructor). If we had more types in the program, corresponding size functions will be generated. These upper bound functions can be easily integrated into the simulation process by means of annotations. For example, consider the following main method that invokes sum and note the Cost annotation at Line 22:

```
20 Unit main() {
21   List<Int> l = Cons(1, Cons(2, Nil));
22   [Cost: ub_sum(l)] this ! sum(l);
23 }
```

Now when reaching Line 22, if the simulator needs (an upper bound on) the cost of the corresponding call to sum it can simply execute the call given in the corresponding Cost annotation, namely ub_sum(l).

Bibliography

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, May 1995.

[2] P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter Schmitt, and Mattias Ulbrich, editors. *The KeY Book: Deductive Software Verification in Practice*, volume 10001 of *LNCS*. Springer-Verlag, to appear, 2016.

[4] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer-Verlag, 2014.

[5] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In Erika Ábrahám and Catuscia Palamidessi, editors, *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE 2014)*, volume 8461 of *Lecture Notes in Computer Science*, pages 49–65. Springer-Verlag, 2014.

[6] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Transactions on Computational Logic*, 17(2):11:1–11:39, 2016.

[7] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *25th International Conference on Compiler Construction (CC'16)*, pages 269–270. ACM, 2016.

[8] Bernhard Beckert and Reiner Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1):20–29, Jan.–Feb. 2014.

[9] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[10] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A fresh approach to learning register automata. In Marie-Pierre Béal and Olivier Carton, editors, *DLT'13*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

[11] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[12] M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163. IEEE, 2013.

[13] F.S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, and G. Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In *Proc. of Formal Aspects of Component Software - 9th International Workshop, FACS 2012*, volume 7684 of *Lecture Notes in Computer Science*, pages 110–127. Springer-Verlag, 2012.

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

[15] Initial User Requirements, January 2014. Deliverable D4.1 of project FP7-610582 (ENVISAGE), available at `http://www.envisage-project.eu`.

[16] Verification, March 2015. Deliverable D3.2 of project FP7-610582 (ENVISAGE), available at `http://www.envisage-project.eu`.

[17] Assurance of the FRH Case Study, September 2016. Deliverable D4.3.3 of project FP7-610582 (ENVISAGE), available at `http://www.envisage-project.eu`.

[18] Test Case Generation, March 2016. Deliverable D3.5 of project FP7-610582 (ENVISAGE), available at `http://www.envisage-project.eu`.

[19] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *ESOP'12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[20] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer-Verlag, 2015.

[21] Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.

[22] Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In Michael Butler et al, editor, *In Proceedings of the 17th International conference on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2015.

[23] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. of POPL'05*, pages 110–121. ACM, 2005.

[24] 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.

[25] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/Guarantee Reasoning for Asynchronous Programs. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 483–496, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[27] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*, pages 394–411. Springer, 2013.

[28] 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.

[29] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *CoRR*, abs/1511.04926, 2015.

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

[31] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL'08*, pages 273–284. ACM, 2008.

[32] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[33] Eduard Kamburjan. Session Types for ABS. Technical report, TU Darmstadt, 2016. www.se.tu-darmstadt.de/publications/details/?tx_bibtex_pi1[pub_id]=tud-cs-2016-0179.

[34] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional verification on actor-based concurrent systems. In Luca Aceto and Anna Ingolfsdottir, editors, *Proc. 27th Nordic Workshop on Programming Theory (NWPT), Reykjavik University, Iceland*. Icelandic Centre of Excellence in Theoretical Computer Science (ICE-TCS) and the School of Computer Science at Reykjavik University, 2015.

[35] 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.

[36] Stephen P. Masticola and Barbara G. Ryder. A model of ada programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, USA, May 20-21, 1991*, pages 97–107. ACM, 1991.

[37] Bertrand Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446, 1998.

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

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

[40] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[41] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[42] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[43] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

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

[45] 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.

Appendix A

Session Types for ABS

Session-Based Compositional Analysis for Actor-Based Languages Using Futures *

Eduard Kamburjan[1], Crystal Chang Din[2], and Tzu-Chun Chen[1]

[1] Department of Computer Science, TU Darmstadt, Germany
[2] Department of Informatics, University of Oslo, Norway
kamburjan@cs.tu-darmstadt.de, crystald@ifi.uio.no,
tc.chen@dsp.tu-darmstadt.de

Abstract. This paper proposes a simple yet concise framework to statically verify communication correctness in a concurrency model using futures. We consider the concurrency model of the core ABS language, which supports actor-style asynchronous communication using futures and cooperative scheduling. We provide a type discipline based on session types, which gives a high-level abstraction for structured interactions. By using it we statically verify if the local implementations comply with the communication correctness. We extend core ABS with sessions and annotations to express scheduling policies based on required communication ordering. The annotation is statically checked against the session automata derived from the session types.

1 Introduction

While distributed and concurrent systems are the pillars of modern IT infrastructures, it is non-trivial to model asynchronous interactions and statically guarantee communication correctness of such systems. This challenge motivates us to bring a compositional analysis framework, which models and locally verifies the behaviors of each distributed endpoints (i.e. components) from the specification of their global interactions. For modeling, we focus on core ABS [10,15], an object-oriented actor-based language designed to model distributed and concurrent systems with asynchronous communications. For verification, we establish a hybrid analysis, which statically type checks local objects' behaviors and, at the same time ensures that local schedulers obey to specified policies during runtime. We apply *session types* [12, 21] to type interactions by abstracting structured communications as a global specification, and then automatically generating local specifications from the global one to locally type check endpoint behaviors.

The distinguishing features of the core ABS concurrency model are (1) *cooperative scheduling*, where methods explicitly control internal interleavings by explicit scheduling points, and (2) the usage of *futures* [11], which decouple the process invoking a method and the process reading the returned value. By sharing future identities, the caller enables other objects to wait for the same method

* Every author contributed to this paper equally.

results. Note that core ABS does not use channels. Communication between processes is restricted to method calls and return values.

The order of operations on futures is fixed: First a fresh future identity is created upon the caller invoking a method on the callee, then the callee starts the method execution. After method termination, the callee sends the result to the future, i.e. future is resolved. Finally, any object which can access the future can read the value from this future. Our session-based type system ensures that the specification respects this order. We have two kinds of communications: Caller invoking a method at a remote callee, and callee returning values via a future to those who know that future. The later is non-trivial since several endpoints can read more than once from the same resolved future at any time.

To the best of our knowledge, it is the first time that session types are considered for typing the concurrency model of core ABS. Our contributions include: (1) extending core ABS with sessions (SABS for short) by giving special annotations to specify the order of interactions among concurrent class instances, (2) establishing a session-based type system, and (3) generating session automata [2] from session types to represent scheduling policies and typestate [20]. To capture the interactions among objects, which are running several processes, we introduce a *two-fold notion of local types*: Object types defining behaviors (i.e. including scheduling behavior) among class instances, while method types defining behaviors that processes should follow.

Outline: Section 2 gives a motivating example which is used in the rest of the paper. Section 3 introduces the concurrency model of SABS. Section 4 defines session types for SABS (ABS-ST for short), while Section 5 gives a type system. Section 6 introduces session automata which are used to verify behaviors of schedulers. Section 7 gives the related works, while section 8 concludes our work.

2 Motivating Example: A Grading System

Consider a service, called *grading system*, which offers an expensive computation on sensitive data, e.g. automatic evaluation of exams. This service consists of three endpoints: A computation server, denoted by c, and a service desk, denoted by d, where a student, denoted by s, can request their grades. The protocol is as follows: Once c finishes calculating the grades, it sends a *publish* message containing the grades to d and an announcement, *announce* message, to a student. It is not desirable that d starts a new communication with c because c may be already computing the next exams; it is also not desirable that d communicates to s without any request from s.

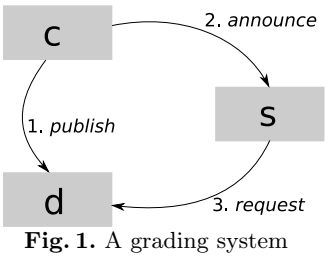


Fig. 1. A grading system

If a student requests his/her grades before the service desk receives the grades from c by *publish*, the scheduler of d must postpone the process of *request* until *publish* has been executed and terminated. This is not possible for core ABS, because a scheduler in core ABS cannot be idle while waiting for a specific message when the process queue is non-empty. Thus we propose an extension of

core ABS to ensure that the endpoints and their local schedulers behave well to the specified communication order.

3 The Session-based ABS Language (SABS)

This section introduces the concept of *session* to core ABS [10, 15]. The extended language is called session-based ABS, SABS in short. The goal of this extension is to equip the language's compiler with the ability to statically ensure communication correctness for applications written in core ABS.

3.1 Syntax and the concurrency model of core ABS

The SABS language provides a combination of algebraic datatype, functional sublanguage, and a simple imperative object-oriented language. The former two kinds are kept the same in SABS as in core ABS. The imperative object-oriented layer is extended. The syntax of SABS can be found in Fig. 2, in which the new language extension is highlighted and will be explained in Section 3.2.

Syntactic categories	*Definitions*
T in Ground Type	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle \quad B ::= \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{String} \mid \mathsf{Unit} \mid \cdots$
B in Basic Type	$A ::= N \mid T \mid D\langle\overline{A}\rangle$
A in Generic Type	$Dd ::= \texttt{data}\ D[\langle\overline{A}\rangle] = Cons[\overline{\mid Cons}]; \qquad Cons ::= Co[(\overline{A})]$
N in Names	$F ::= \texttt{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\,\overline{x}) = e;$
e in Expression	$e ::= b \mid x \mid t \mid \texttt{this} \mid \texttt{destiny} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{case}\ e\ \{\overline{br}\}$
x in Variable	$t ::= Co[(\overline{t})] \mid \texttt{null} \quad br ::= p \Rightarrow e; \quad p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$
t in Ground Term	$P ::= \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x};\ s\}$
br in Branch	$IF ::= \texttt{interface}\ I\ \{\ \overline{Sg}\ \}$ $\qquad Sg ::= T\ m\ (\overline{T}\,\overline{x})$
p in Pattern	$CL ::= [[a]]\texttt{class}\ C\ [(\overline{T}\,\overline{x})]\ [\texttt{implements}\ \overline{I}]\ \{\ \overline{T}\,\overline{x};\ \overline{M}\}$
C, I, m in Names	$M ::= Sg\ \{\ \overline{T}\,\overline{x};\ s\ \}$
L in Local Types	$a ::= \texttt{Scheduler: L} \quad ptc ::= \texttt{Protocol: G} \quad join ::= \texttt{Ses}\text{: String}$
S in Session IDs	$rhs ::= e \mid e!m(\overline{e}) \mid e.\texttt{get}$
G in Global Types	$s ::= s; s \mid x = rhs \mid \texttt{skip} \mid \texttt{return}\ e$
s in Statement	$\mid \texttt{await}\ e? \mid \texttt{if}\ b\ \{s\}\ [\texttt{else}\ \{s\}] \mid \texttt{while}\ b\ \{s\} \mid \texttt{case}\ e\{\overline{p \Rightarrow s;}\}$
b in Bool Expression	$\mid [ptc]\ \mathrm{x} = \texttt{new}\ Session(\mathrm{e}) \mid [join]\ x = \texttt{new}\ C[(\overline{e})]$

Fig. 2. SABS syntax. Terms $\overline{\ }$ denote possibly empty lists over corresponding syntactic categories, and [] optional elements. The highlighted ones are the new syntax added to core ABS.

A SABS model, denoted by P, defines datatypes $\overline{Dd}$, functions $\overline{F}$, interfaces $\overline{IF}$, classes $\overline{CL}$, and main block $\{\overline{T}\ \overline{x};\ \mathrm{s}\}$ to configure the initial state. In datatype declarations Dd, an abstract datatype D has at least one constructor $Cons$, which has a name Co and a list of generic types $\overline{A}$ for its arguments. A future of built-in type $\texttt{Fut}\langle T\rangle$ expresses that the value stored in the future is of type T. Function declarations F consist of a return type A, a function name fn, an optional list of parameters of types $\overline{A}$, a list of variable declarations $\overline{x}$ of types $\overline{A}$, and an expression e. Expressions e include boolean expressions b, variables x, (ground) terms t, the self-identifier **this**, the return address **destiny** of the method activation, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$ where br is a branch. An interface $\overline{IF}$

has a name I and method signatures $\overline{Sg}$. A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. A class CL has a name C, the fields $\overline{x}$ of type $\overline{T}$ for formal parameters and state variables, implemented interfaces $\overline{I}$ and methods $\overline{M}$. The right-hand side expressions rhs include (pure) expressions e, asynchronous remote method invocation $e!m(\overline{e})$, and future fetching expression $e.\mathbf{get}$. Statements s include sequential composition, assignment, session creation, object creation, guarding statement $\mathbf{await}\ e?$, **if**, **while**, branching, **skip** and **return** statement.

The concurrency model In SABS each object has one scheduler and one processor. It is possible to have more than one processes on an object, but at most one process is executed by the processor on an object at a time. For a method call, a fresh future identity, say f, is generated by the caller upon sending an asynchronous remote method invocation to the callee. A future can be seen as a placeholder for the method result. The callee creates a new process for the receiving call. If the processor of the callee is busy while the new message arrives, the created process will be put into the process pool and can later be chosen for execution by the scheduler. Upon method termination, the callee returns the result to f, i.e. f is resolved. Any object sharing the identity f can read the value from f by executing $f.\mathbf{get}$. This statement blocks the current process until f is resolved and then returns the value. Since execution control is not transferred between objects and there is no direct access from one object to the fields of other objects, each object can be analyzed individually. SABS supports cooperative scheduling. Each object relies on its scheduler to select a process for execution at the explicit scheduling points, which can be upon termination of object initialization, at **await** statement, and upon method termination. When a process execution encounters statement $\mathbf{await}\ f?$, if the future f is not resolved yet, the processor is released and the current process is suspended and put into the process pool. Then the processor is idle and the scheduler chooses a process from the pool for execution based on a scheduling policy (i.e. specified by a local specification). We say the chosen process from the pool is reactivated.

3.2 New language extension

SABS provides a set of new features in order to guide the scheduler to select the intended process for execution according to the required interaction ordering. In Fig. 2, the statement **[Protocol : G]** $x = \mathbf{new}\ Session(e)$ creates a new session with a fresh session id stored in x. The parameter e is the session name of type *String*. The annotation **[Protocol : G]** describes the global communication specification $\mathbf{G}$, which will be formalized in Section. 4.1, that the newly created session should obey. The statement **[Ses** : S**]** $x = \mathbf{new}\ C[(\overline{e})]$ creates a new object with a fresh object id stored in x. The annotation **[Ses** : S**]** specifies that the newly created object belongs to session S. Each object can belong to at most one session. The annotation **[Scheduler : L]** is optional and can be added in front of the class declarations. It provides the local communication specification $\mathbf{L}$ to guide the scheduler of the current object.

The SABS implementation for the grading example in Section 2 is in Fig. 3, in which we create a new session ses named Service. Type gradingSystem de-

```
1  interface ServiceDesk{ Unit publish(Int g); Int request(); }
2  interface CompServer{ Unit pubGrd(ServiceDesk d, Student s, Int g); }
3  interface Student{ Unit announce(ServiceDesk d); }
4
5  [ Scheduler: c?_f publish.s?_f' request ] // local protocol for SD
6  class SD(CompServer c, Student s) implements ServiceDesk{
7          Int grade = 0;
8          Unit publish(Int g){grade = g;}
9          Int request(){return grade;}}
10
11 class CS implements CompServer{
12         Unit pubGrd(ServiceDesk d, Student s, Int g){
13                 Fut<Unit> f = d!publish(g,this);
14                 Fut<Unit> f' = s!announce(d,this);
15         }}
16
17 class S implements Student{
18         Int grade = 0;
19         Unit announce(ServiceDesk d){
20                 Fut<Int> f'' = d!request(this);
21                 grade = f''.get;
22         }}
23
24 { [ Protocol: gradingSystem ] Ses ses = new Session("Service");
25   [ Ses: ses ] CompServer c = new CS;
26   [ Ses: ses ] Student s = new S;
27   [ Ses: ses ] ServiceDesk d = new SD(c,s);
28   Fut<Unit> f0 = c!pubGrd(d,s,85); }
```

Fig. 3. The ABS implementation for the example in Section. 2.

fines this session's global communication specification (introduced later in Section 4.1). Computer server c, student s and service desk d all belong to the same session ses. The scheduling policy for the service desk is represented by a local specification $\mathsf{c}?_f\mathit{publish}.\mathsf{s}?_{f'}\mathit{request}$ (introduced later in Section 4.2), which specifies the method request invoked by the student s can only be executed after the execution of method publish invoked by the computer server c.

4 Compositional Analysis Based on Session Types

Based on the approach of compositional analysis and the theory of session types [12], we introduce the ABS-ST (*ABS Session Types*) framework: Each object (i.e. component) is statically checked against its local specification, which are projected from a global specification, specifying the overall interactions among objects (i.e. composes objects). As multiparty session types type interactions consisting of simple sending and receiving actions among multiple processes, ABS-ST type interactions consisting of asynchronous remote method calls, scheduling, and futures among *objects*. This work extends [16], which contains proofs, full examples and definitions.

4.1 Global Types

Global types, denoted by $\mathbf{G}$, define global communication specifications within a closed system of objects. Contrary to session types [12, 21], we do not specify the datatype of a message since the message is a method call or a method return and every method in SABS has a fixed signature. The syntax of $\mathbf{G}$ is defined:

Definition 1. Let $\mathbf{p}, \mathbf{q}$ range over objects, denoted by Ob, f over futures, m over method names and $\mathtt{C}$ over all constructors of all abstract datatypes.

$$\mathbf{G} ::= \mathbf{0} \xrightarrow{f} \mathbf{q}{:}m \mid \mathbf{G}.\mathbf{g} \qquad \mathbf{g} ::= \mathbf{p} \xrightarrow{f} \mathbf{q}{:}m.\mathbf{g} \mid \mathbf{p}{\downarrow}f{:}(\mathtt{C}).\mathbf{g} \mid \mathbf{p}{\uparrow}f{:}(\mathtt{C}).\mathbf{g} \mid \mathsf{Rel}(\mathbf{p}, f).\mathbf{g} \mid \mathbf{p}\{\mathbf{g}_j\}_{j\in J} \mid \mathsf{end} \mid \mathbf{g}^*$$

Initialization $\mathbf{0} \xrightarrow{f} \mathbf{q}{:}m$ starts interactions from the *main block* invoking object $\mathbf{q}$, e.g. we write $\mathbf{0} \xrightarrow{f_0} \mathbf{c}{:}\mathit{pubGrd}$ to specify the code in the *main block* in Fig. 3. We use . for sequential composition and write $\mathbf{G}.\mathbf{g}$ to mean interaction(s) $\mathbf{g}$ follows $\mathbf{G}$. Interaction $\mathbf{p} \xrightarrow{f} \mathbf{q}{:}m$ models a remote call, where object $\mathbf{p}$ asynchronously calls method m at object $\mathbf{q}$ via future f, and then $\mathbf{q}$ creates a new process for this method call. The resolving type $\mathbf{p}{\downarrow}f{:}(\mathtt{C})$ models object $\mathbf{p}$ resolving the future f. If the method has an algebraic datatype as its return type, then the return value has $\mathtt{C}$ as its outermost constructor; otherwise we simply write $\mathbf{p}{\downarrow}f$. The fetching type $\mathbf{p}{\uparrow}f{:}(\mathtt{C})$ models object $\mathbf{p}$ reading the future f. The usage of $\mathtt{C}$ here is similar to the one in $\mathbf{p}{\downarrow}f{:}(\mathtt{C})$. The releasing type $\mathsf{Rel}(\mathbf{p}, f)$ models $\mathbf{p}$ which releases the control until future f has been resolved. This type corresponds to **await** f? statement in SABS. The example below shows how $\mathsf{Rel}(\mathbf{p}, f)$ works:

Example 1. Consider $\mathbf{G}_{\text{release}} = \mathbf{0} \xrightarrow{f_0} \mathbf{a}{:}m_0.\mathbf{a} \xrightarrow{f_1} \mathbf{b}{:}m_1.\mathbf{b} \xrightarrow{f_2} \mathbf{a}{:}m_2.\mathbf{g}$. It does not specify the usage of futures correctly: At the moment $\mathbf{b}$ makes a remote call on m_2 at $\mathbf{a}$, the process computing f_0 is still active at $\mathbf{a}$. We shall revise it to

$$\mathbf{G}'_{\text{release}} = \mathbf{0} \xrightarrow{f_0} \mathbf{a}{:}m_0.\mathbf{a} \xrightarrow{f_1} \mathbf{b}{:}m_1.\mathsf{Rel}(\mathbf{a}, f_1).\mathbf{b} \xrightarrow{f_2} \mathbf{a}{:}m_2.\mathbf{g}$$

Here $\mathbf{a}$ suspends its first process computing f_0 until f_1 has been resolved; during this period, $\mathbf{a}$ can execute the call on m_2.

The branching type $\mathbf{p}\{\mathbf{g}_j\}_{j\in J}$ expresses that as $\mathbf{p}$ selects the jth branch, $\mathbf{g}_j$ guides the continuing interactions. The type end means termination.

Note that only a *self-contained* $\mathbf{g}$ can be repeatedly used. We say $\mathbf{g}$ is *self-contained* if (1) wherever there is a remote call or releasing, there is a corresponding resolving and visa versa; and (2) it contains no end, and (3) every repeated type within it is also *self-contained*. We say $A \in \mathbf{g}$ if A appears in $\mathbf{g}$ and $A \in \mathbf{G}$ if $A \in \mathbf{g}$ for some $\mathbf{g} \in \mathbf{G}$. E.g., we have $\mathbf{q} \xrightarrow{f} \mathbf{p} : m \in \mathbf{0} \xrightarrow{f_0} \mathbf{q} : m.\mathbf{q} \xrightarrow{f} \mathbf{p} : m$ and $\mathbf{q}\{\mathbf{g}_2\} \in \mathbf{q}\{\mathbf{g}_j\}_{j\in\{1,2,3\}}$ and its negation means the inverse. A future f is *introduced in* $\mathbf{g}$ (or $\mathbf{G}$) if $\mathbf{p} \xrightarrow{f} \mathbf{q} \in \mathbf{g}$ (or $\mathbf{G}$).

Now we define type $\mathbf{g}^* = \mathit{fresh}(\mathbf{g}).\mathbf{g}^*$ (in case $\mathit{fresh}(\mathbf{g})$ is a branching, we append $\mathbf{g}^*$ to the end of every branch), meaning finite repetition of a *self-contained*

$\mathbf{g}$ by giving every repetition fresh future names:

$$\mathit{fresh}(\mathbf{g}) = \begin{cases} \mathbf{g}\{f_1'/f_1\}...\{f_n'/f_n\} & \text{if } f_1,...,f_n \text{ are } \textit{introduced in } \mathbf{g} \text{ and } f_1', .., f_n' \text{ fresh} \\ \mathbf{p}\{\mathit{fresh}(\mathbf{g}_j)\}_{j\in J} & \text{if } \mathbf{p}\{\mathbf{g}_j\}_{j\in J} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

In other words, we need to keep *linearity* of futures for every iterations.

Example 2. We show how the global type gradingSystem, used in the code of Fig. 3, represents the grading system discussed in Section 2:

$$\begin{aligned} \mathsf{gradingSystem} = \ & \mathbf{0} \xrightarrow{f_0} \mathbf{c}:\mathit{pubGrd}.\mathbf{c} \xrightarrow{f} \mathbf{d}:\mathit{publish}.\mathbf{d}\downarrow f.\mathbf{c} \xrightarrow{f'} \mathbf{s}:\mathit{announce}. \\ & \mathbf{s} \xrightarrow{f''} \mathbf{d}:\mathit{request}.\mathbf{d}\downarrow f''.\mathbf{s}\uparrow f''.\mathbf{s}\downarrow f'.\mathbf{c}\downarrow f_0.\mathsf{end} \end{aligned}$$

The session is started by a call on $\mathbf{c}.\mathit{pubGrd}$, while other objects are inactive at the moment. After the call $\mathbf{c} \xrightarrow{f} \mathbf{d}:\mathit{publish}$, the service desk $\mathbf{d}$ is active at computing f in a process running *publish*. We position $\mathbf{d}\downarrow f$ there to specify that $\mathbf{d}$ must resolve f after it is called by $\mathbf{c}$ and before it is called by $\mathbf{s}$ (i.e. $\mathbf{s} \xrightarrow{f''} \mathbf{d}:\mathit{request}$). For $\mathbf{c}$, it can have a second remote call $\mathbf{c} \xrightarrow{f'} \mathbf{s}:\mathit{announce}$ after its first call. Thus in this case it is no harm to move $\mathbf{d}\downarrow f$ right after $\mathbf{c} \xrightarrow{f'} \mathbf{s}:\mathit{announce}$. As $\mathbf{d}$ is called by $\mathbf{s}$, $\mathbf{d}$ can start computing f'' in a process running *request* only after $\mathbf{d}\downarrow f$, which means the process computing *publish* has terminated. $\mathbf{s}$ will fetch the result by $\mathbf{s}\uparrow f''$ after $\mathbf{d}$ resolves f''; then $\mathbf{s}$ resolves f'. Note that, since $\mathbf{c}$ does not need to get any response from $\mathbf{d}$ nor $\mathbf{s}$, $\mathbf{c}$ simply finishes the session by $\mathbf{c}\downarrow f_0$. The end is there to ensure all processes in the session terminate. The valid use of futures is examined during generating object types from a global type, a procedure introduced in Section 4.3. If $\mathbf{s}\uparrow f''$ is specified before $\mathbf{d}\downarrow f''$, the projection procedure will return *undefined* since f'' can not be read before being resolved.

4.2 Local Types

Besides global types, to statically check code, we define local types, which describe local specifications at object level. The syntax of local types is defined:

Definition 2.

$$\begin{aligned} \mathbf{L} \quad ::= \quad & \mathbf{p}!_f m.\mathbf{L} \mid \mathbf{p}?_f m.\mathbf{L} \mid \mathsf{Put}\ f:(\mathtt{C}).\mathbf{L} \mid \mathsf{Get}\ f:(\mathtt{C}).\mathbf{L} \mid \mathsf{Await}(f,f').\mathbf{L} \\ & \mid \mathsf{React}(f).\mathbf{L} \mid \oplus\{\mathbf{L}_j\}_{j\in J} \mid \&_f\{\mathbf{L}_j\}_{j\in J} \mid \mathbf{L}^*.\mathbf{L} \mid \mathsf{skip}.\mathbf{L} \mid \mathsf{end} \end{aligned}$$

We use . to denote sequential composition. The type $\mathbf{p}!_f m$ denotes a sending action via an asynchronous remote call on method m at endpoint $\mathbf{p}$. The type $\mathbf{p}?_f m$ denotes a receiving action which starts a new process computing f by executing method m after a call from $\mathbf{p}$. The resolving $\mathsf{Put}\ f:(\mathtt{C})$ and fetching $\mathsf{Get}\ f:(\mathtt{C})$ have the same intuitive meaning as their global counterparts. The suspension $\mathsf{Await}(f,f')$ means that the process computing f suspends its action

until future f' is resolved. The reactivation $\mathsf{React}(f)$ means the process continues the execution with f. The choice operator $\oplus$ in $\oplus\{\mathbf{L}_j\}_{j\in J}$ denotes that the currently active process selects a branch to continue. The offer operator $\&_f$ in $\&_f\{\mathbf{L}_j\}_{j\in J}$ denotes that the object offers branches $\{\mathbf{L}_j\}_{j\in J}$ when f is resolved. The type skip denotes no action and we say $\mathbf{L}.\mathsf{skip} \equiv \mathbf{L} \equiv \mathsf{skip}.\mathbf{L}$.

In ABS-ST the communication happens among processes in different objects. We list three kinds of local types:

- *A method type* describes the execution of a single process on a particular future f. It has the following attributes: (1) Its first action is $\mathbf{p}?_f m$ for some $\mathbf{p}$, m, f, and (2) if it has a branching type, the final action in every branch is $\mathsf{Put}\ f\!:\!(\mathtt{C})$ for some $\mathtt{C}, f$, and (3) it contains no further resolving action or receiving action, and (4) it contains no end.
- *An object type* is a type which is not a method type.
- *A condensed type*, denoted by $\hat{\mathbf{L}}$, where $\mathbf{L}$ is an object type, replaces every action, except receiving and reactivation actions, in $\mathbf{L}$ with skip.

Example 3. Consider object $\mathbf{d}$ in the *grading system* in Section 2. Its method type on future f, which is used for calling method *publish*, is $\mathbf{c}?_f \mathit{publish}.\mathsf{Put}\ f$. Its object type is $\mathbf{L} = \mathbf{c}?_f \mathit{publish}.\mathsf{Put}\ f.\mathbf{s}?_{f''} \mathit{request}.\mathsf{Put}\ f''.\mathsf{end}$, and its condensed type is $\hat{\mathbf{L}} = \mathbf{c}?_f \mathit{publish}.\mathsf{skip}.\mathbf{s}?_{f''} \mathit{request}.\mathsf{skip}.\mathsf{skip} \equiv \mathbf{c}?_f \mathit{publish}.\mathbf{s}?_{f''} \mathit{request}$.

4.3 Projection

Projection is the procedure to derive local types of endpoints from a global type. Since in SABS data is sent between different objects by active processes, the projection rules have two levels: (1) Projecting a global type on objects and resulting *object types* and (2) projecting *object types* on a *future* and resulting *method types*, which type the behavior of process for computing the target future.

4.4 Projecting a Global Type to Local Types

We say a global type is *projectable* if every projection on every of its participants is defined and every future is introduced exactly once (i.e. linearity). A projectable global type implies that the futures appear in it are located correctly across multiple objects; thus the object types gained from it ensure the correct usage of futures.

We define $\mathit{pre}(\mathbf{G}, \mathbf{G}')$ as the set of prefixes of $\mathbf{G}$:

$$\mathit{pre}(\mathbf{G}, \mathbf{G}') = \{\mathbf{G}'' \mid \mathbf{G} \in \mathbf{G}' \text{ implies } \mathbf{G}''.\mathbf{G} \in \mathbf{G}'\}$$

and that a future f *introduced in* $\mathbf{G}'$ is *active* on object $\mathbf{o}$ in $\mathbf{G}$ iff (if and only if):

$$\begin{aligned}&(\mathbf{p} \xrightarrow{f} \mathbf{o} \in \mathit{pre}(\mathbf{G}, \mathbf{G}')) \wedge (\mathbf{o}{\downarrow} f\!:\!(\mathtt{C}) \notin \mathit{pre}(\mathbf{G}, \mathbf{G}'))\\ \wedge \big(&(\mathsf{Rel}(\mathbf{p}, f') \in \mathit{pre}(\mathbf{G}, \mathbf{G}') \wedge\ f \text{ active in } \mathsf{Rel}(\mathbf{p}, f')) \rightarrow \mathbf{o}{\downarrow} f' \in \mathit{pre}(\mathbf{G}, \mathbf{G}')\big)\end{aligned}$$

The first conjunct captures that after $\mathbf{p} \xrightarrow{f} \mathbf{o}$, f becomes active on $\mathbf{o}$, while after $\mathbf{o}{\downarrow} f\!:\!(\mathtt{C})$, f becomes inactive on $\mathbf{o}$; the second conjunct captures that if f has been suspended on f' (i.e. $\mathsf{Rel}(\mathbf{p}, f') \in \mathit{pre}(\mathbf{G}, \mathbf{G}') \wedge f$ active in $\mathsf{Rel}(\mathbf{p}, f')$), then f must have been reactivated by resolving f' (i.e. $\mathbf{o}{\downarrow} f' \in \mathit{pre}(\mathbf{G}, \mathbf{G}')$).

Projection from global types to object types

$$pj(\mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m, \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathbf{q}!_f m & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (f \text{ is fresh}) \\ \mathbf{p}?_f m & \text{if } (\mathbf{o} = \mathbf{q}) \wedge (f \text{ is fresh}) \wedge (\mathbf{q} \text{ is inactive}) \\ \mathsf{skip} & \text{if } (\mathbf{o} \neq \mathbf{q} \wedge \mathbf{o} \neq \mathbf{p}) \wedge (f \text{ is fresh}) \end{cases}$$

$$pj(\mathbf{p}{\downarrow}f\!:\!(\mathtt{C}), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Put}\ f\!:\!(\mathtt{C}) & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (f \text{ is active on } \mathbf{p}) \\ \mathsf{React}\ f' & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\mathsf{Rel}(\mathbf{o}, f) \in pre(\mathbf{p}{\downarrow}f\!:\!(\mathtt{C}), \mathbf{G})) \\ & \quad \wedge (\ f' \text{ is active on } \mathbf{o} \text{ in } \mathsf{Rel}(\mathbf{o}, f)) \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$pj(\mathbf{p}{\uparrow}f\!:\!(\mathtt{C}), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Get}\ f\!:\!(\mathtt{C}) & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (\mathbf{q}{\downarrow}f\!:\!(\mathtt{C}) \in pre(\mathbf{p}{\uparrow}f\!:\!(\mathtt{C}), \mathbf{G})) \\ \mathsf{skip} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\mathbf{q}{\downarrow}f\!:\!(\mathtt{C}) \in pre(\mathbf{p}{\uparrow}f\!:\!(\mathtt{C}), \mathbf{G})) \end{cases}$$

$$pj(\mathsf{Rel}(\mathbf{p}, f), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Await}(f', f) & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (f' \text{ is active on } \mathbf{p}) \\ & \quad \wedge (\nexists \mathbf{p}'.\ \ \mathbf{p}'{\downarrow}f \in pre(\mathsf{Rel}(\mathbf{p}, f), \mathbf{G})) \\ \mathsf{skip} & \text{if} (\mathbf{o} \neq \mathbf{p}) \end{cases}$$

$$pj(\mathbf{p}\{\mathbf{g}_j\}_{j\in J}, \mathbf{o})_{\mathbf{G}} = \begin{cases} \oplus\{pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}}\}_{j\in J} & \text{if } (\mathbf{o} = \mathbf{p}) \\ \&_f\{pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}}\}_{j\in J} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (f \text{ is active on } \mathbf{o}) \\ \mathbf{L} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\forall j \in J.\ pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}} = \mathbf{L}) \end{cases}$$

$$pj(\mathsf{end}, \mathbf{o})_{\mathbf{G}} = \mathsf{end} \qquad pj(\mathbf{g}^*, \mathbf{o})_{\mathbf{G}} = (pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}})^* \text{ if } \mathbf{g} \text{ is self-contained}$$

$$pj((\mathbf{G}'.\mathbf{g}), \mathbf{o})_{\mathbf{G}} = \begin{cases} pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}} & \text{if } (pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}} = \mathsf{skip}) \\ pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}}.pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}} & \text{otherwise} \end{cases}$$

Projection from object types to method types

$$\mathsf{SIMPLE} = \{\mathbf{p}!_{f'}m, \mathbf{p}?_f m, \mathsf{Put}\ f\!:\!(\mathtt{C}), \mathsf{Get}\ f'\!:\!(\mathtt{C}), \mathsf{Await}(f, f')\}$$

$$pjm(\mathbf{L}, f) = \begin{cases} \mathbf{L} & \text{if } (\mathbf{L} \in \mathsf{SIMPLE}) \wedge (f \text{ is active}) \\ \mathsf{skip} & \text{if } (\mathbf{L} \in \mathsf{SIMPLE}) \wedge (\text{some other future is active}) \\ \mathsf{skip} & \text{if } (\mathbf{L} = \mathsf{React}(f)) \vee (\mathbf{L} = \mathsf{end}) \\ pjm(\mathbf{L}_1, f).pjm(\mathbf{L}_2, f) & \text{if } (\mathbf{L} = \mathbf{L}_1.\mathbf{L}_2) \\ pjm(\mathbf{L}', f) & \text{if } (\mathbf{L} = (\mathbf{L}')^*) \wedge (\mathbf{p} \xrightarrow{f} \mathbf{q} \in \mathbf{L}') \\ pjm(\mathbf{L}', f)^* & \text{if } (\mathbf{L} = (\mathbf{L}')^*) \wedge (\mathbf{p} \xrightarrow{f} \mathbf{q} \notin \mathbf{L}') \\ pjm(\mathbf{L}_j, f) & \text{if } ((\mathbf{L} = \&_{f'}\{\mathbf{L}_j\}_{j\in J}) \vee (\mathbf{L} = \oplus\{\mathbf{L}_j\}_{j\in J})) \\ & \quad \wedge (f \neq f') \wedge (\mathbf{p} \xrightarrow{f} \mathbf{o} \in \mathbf{L}_j) \\ \oplus\{pjm(\mathbf{L}_j, f)\}_{j\in J} & \text{if } (\mathbf{L} = \oplus\{\mathbf{L}_j\}_{j\in J}) \\ \&_{f'}\{pjm(\mathbf{L}_j, f)\}_{j\in J} & \text{if } (\mathbf{L} = \&_{f'}\{\mathbf{L}_j\}_{j\in J}) \wedge (\mathbf{L}_j \text{ are distinct}) \end{cases}$$

Fig. 4. Projection rules

Fig. 4 defines the projection rules as a function $pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}}$ projecting $\mathbf{g}$ to object $\mathbf{o}$, where $\mathbf{g} \in \mathbf{G}$. We write $pj(\mathbf{G}, \mathbf{o})_{\mathbf{G}} = \mathbf{G} \upharpoonright \mathbf{o}$. The side-conditions verify the defined cases, where the futures are used correctly; others are undefined.

The interaction type projects a sending action on the caller side and a receiving action on the callee side. A resolving type gives an action for resolving f on the corresponding object, and generates a reactivation for every objects who are waiting for f; for others, it gives skip. A fetching type gives an action for fetching the result from f on the corresponding object and gives skip for others. Its side-condition ensures that a future is resolved before fetching it. This must be checked at a global level because resolving and fetching take place in different objects. A releasing type gives suspension for the corresponding object and gives skip for others. Its side-condition ensures that the releasing object does not have any other future waiting for the same resolving. A branching type gives a choice type for the active side and an offer for every one that either receives one of the calls invoked by the object making the choice or reads from the active future. For other objects, each branch should have the same behavior so that those objects always know how to proceed no matter which branch was selected. Termination type gives end, which means every future has been resolved and all objects are inactive. The repetition and concatenation are propagated down.

4.5 Projecting Object Types to Method Types

Fig. 4 also defines the projection of an object type $\mathbf{L}$ to a method type on a future f, denoted by $pjm(\mathbf{L}, f)$. Since the correct usage of futures has been checked when we do $\mathbf{G} \upharpoonright \mathbf{o}$, $pjm(\mathbf{G} \upharpoonright \mathbf{o}, f)$ has ensured the valid usage of futures.

For a sending, receiving, resolving, fetching, suspending, or repetition object type which is active on future f, the projected method type is itself; otherwise the projection gives skip. A choice object type gives a choice method type when it projects on an active future used by the target object, while gives a unique $\mathbf{L}$ when it projects on a future which only appears in one branch. Similarly for the case of offer object type. A reactivation object type gives a method type skip when it projects on any future because the next action after a suspension will always be a reactivation inside a method type. Termination object type also always gives skip for any future because it is not visible. A concatenation object type gives a concatenation method type on any future.

5 Type System

We say the objects involving in a sequence of communications, i.e. a session, satisfy *communication correctness* iff, during the interactions, they always comply with some pre-defined global type. To locally check endpoint implementations and statically ensure communication safety, which is currently not supported by core ABS, we here introduce a type system, which is defined in Fig. 5.

We use Θ as session environments, which contain sessions associating to global types that they follow, with information about the types of participants; we use Γ as shared environments mapping expressions to ground types, and Δ as channel environments mapping channels (composed by a session name and an object) to local types. Note that, channel environments *only exist in the type system*. When we write Θ, Θ', we mean $domain(\Theta) \cap domain(\Theta') = \emptyset$, so as for φ, Γ and Δ. Θ and Γ together store the shared information. For convenience, we define $role(\mathbf{G})$ returning the set of participants in $\mathbf{G}$, $ptypes(C)$ returning the

Θ(Session Environments) ::= $\emptyset \mid \Theta, \{\varphi\}_{\mathbf{s}:\mathbf{G}}$ $\quad$ $\varphi ::= \emptyset \mid \varphi, I : \mathbf{p} \mid \varphi, \mathsf{Any} : \mathbf{0}$

Γ(Shared Environments) ::= $\emptyset \mid \Gamma, e : T$

Δ(Channel Environments) ::= $\emptyset \mid \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}$

(T-New-Session)

$$\frac{\mathbf{G} \text{ is projectable} \quad \mathbf{s} \text{ fresh} \quad \Theta = \{\mathsf{Any} : \mathbf{0}\}_{\mathbf{s}:\mathbf{G}} \quad \Gamma \vdash e : \mathsf{String}}{\Theta, \Gamma \vdash \texttt{[Protocol: G]}\ \mathsf{Ses}\ \mathbf{s} = \texttt{new}\ \mathit{Session}(\mathrm{e}) \rhd \Delta, \mathbf{s}[\mathbf{0}] : \mathbf{G} \upharpoonright \mathbf{0}}$$

(T-New-Join)

$$\frac{\Theta \vdash \{\varphi\}_{\mathbf{s}:\mathbf{G}} \text{ for some } \mathbf{G} \quad \Gamma \vdash \overline{e} : \mathit{ptypes}(C) \quad \Gamma \vdash c : I}{\Theta, \Gamma \vdash \texttt{[Ses: s]}\ \ c = \texttt{new}\ C[(\overline{e})] \rhd \Delta, \mathbf{s}[c] : \emptyset}$$

(T-Scheduler)

$$\frac{\begin{array}{c}\forall I \in \overline{I}.\mathit{implements}(C, I) \quad \Theta, \Gamma[\texttt{this} \mapsto C, \mathit{fields}(C)] \vdash \overline{M} \rhd \Delta, \mathbf{s}[\mathit{obj}(C)] : \emptyset \\ \Theta = \Theta'', \{\varphi\}_{\mathbf{s}:\mathbf{G}} \quad \exists \mathbf{p} \in \mathit{role}(\mathbf{G})\ s.t.\ \mathbf{G} \upharpoonright \mathbf{p} = \mathbf{L} \quad \Theta' = \Theta'', \{\varphi, I : \mathbf{p}\}_{\mathbf{s}:\mathbf{G}}\end{array}}{\Theta', \Gamma \vdash \texttt{[Scheduler: L]}\ \texttt{class}\ C\ [(\overline{T}\ \overline{x})]\ [\texttt{implements}\ \overline{I}]\ \{\ \overline{T}\ \overline{x};\ \overline{M}\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}$$

(T-Send)

$$\frac{\Theta = \Theta', \{\varphi, I : \mathbf{q}\}_{\mathbf{s}:\mathbf{G}} \quad \Gamma \vdash e : I \quad \Gamma \vdash x : f \quad \Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Theta, \Gamma \vdash x = e!m(\overline{e}); s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{q}!_f m.\mathbf{L}}$$

(T-Method)

$$\frac{\begin{array}{c}(\Theta = \Theta', \{\varphi, T : \mathbf{q}\}_{\mathbf{s}:\mathbf{G}}) \wedge ((T = I) \vee (T = \mathsf{Any})) \\ T \in \overline{T} \quad \Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \quad \Gamma'[\texttt{destiny} \mapsto f'] \vdash s \rhd \Delta, \mathit{pjm}(\mathbf{L}, f')\end{array}}{\Theta, \Gamma \vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; s\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathit{pjm}(\mathbf{q}?_f m.\mathbf{L}, f')}$$

(T-Offer)

$$\frac{\Gamma \vdash e : T \quad J = \{1..n\} \quad \forall j \in J. \Gamma \vdash p_j : T \quad \Gamma \vdash s_j \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}_j \quad f \text{ active on } \mathbf{p} \text{ in } \mathbf{L}_j}{\Gamma \vdash \texttt{case}\ e\{p_1 \Rightarrow s_1, ..., p_n \Rightarrow s_n\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \&_f\{\mathbf{L}_j\}_{j \in J}}$$

(T-Choice)

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash p : T \quad J = \{1..n\} \quad k \in J \quad \Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}_k}{\Gamma \vdash \texttt{case}\ e\ \{p \Rightarrow s\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \oplus\{\mathbf{L}_j\}_{j \in J}}$$

(T-Await)

$$\frac{\mathbf{L} = \mathsf{React}(f).\mathbf{L}' \quad f' \text{ inactive on } \mathbf{p} \text{ in } \mathbf{L}' \quad \Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Gamma \vdash \texttt{await}\ e?; s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathsf{Await}(f, f').\mathbf{L}}$$

(T-Skip)

$$\frac{\Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Gamma \vdash \texttt{skip}; s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathsf{skip}.\mathbf{L}}$$

(T-Return)

$$\frac{\Gamma \vdash e : T \quad \Gamma(\texttt{destiny}) = f \quad \Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Gamma \vdash \texttt{return}\ e; s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathsf{Put}\ f : (\mathtt{C}).\mathbf{L}}$$

(T-While)

$$\frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Gamma \vdash \texttt{while}\ b\{s\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}^\star}$$

(T-Get)

$$\frac{\Gamma \vdash s\{e.\mathbf{get}/x\} \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L}}{\Gamma \vdash x = e.\mathbf{get}; s \rhd \Delta, \mathbf{s}[\mathbf{p}] : \mathsf{Get}\ f : (\mathtt{C}).\mathbf{L}}$$

Fig. 5. The type system for the concurrent object level of ABS (Parts: Session-related)

types of parameters of C, $\mathit{implements}(C, I)$ returning true if C can implement interface I, $\mathit{obj}(C)$ returning an instance of class C, and $\mathit{fields}(C)$ returning a shared environment containing attributes of C. We only list the session-related typing rules related. Others are as same as those in core ABS [15].

Rule (T-New-Session) types a session creation by checking if $\mathbf{G}$ in the annotation is projectable (see Section 4.4), the session id $\mathbf{s}$ is fresh, and the type of e is String. If all conditions are satisfied, we create $\Theta = \{\mathsf{Any} : \mathbf{0}\}_{\mathbf{s}:\mathbf{G}}$ to record mappings of types to the participants in $\mathbf{G}$. The first mapping is $\mathsf{Any} : \mathbf{0}$, in which Any types the session initializer. Also, a channel $\mathbf{s}[\mathbf{0}]$ and its type $\mathbf{G} \upharpoonright \mathbf{0}$ is created in shared environments to specify this object playing $\mathbf{0}$ in $\mathbf{G}$. Rule (T-New-Join) types an object creation, which joins session $\mathbf{s}$, which is a name (with type String) of a session. The object creation is valid if $\mathbf{s}$ has been created (i.e. $\Theta \vdash \{\varphi\}_{\mathbf{s}:\mathbf{G}}$) and the type of $\overline{e}$ is $ptypes(C)$.

Rule (T-Scheduler) is the key rule to activate session-based typing. A class with annotation **[Scheduler: L]** is well-typed if its methods are well-typed (this part is as same as the rule (T-Class) in [15]) and, by given the fact that the instance of C has joined session $\mathbf{s}$ (i.e. $\mathbf{s}[obj(C)] : \emptyset$), the local scheduler who specifies the behavior of $obj(C)$ against $\mathbf{L}$ should find $\mathbf{L} = \mathbf{G} \upharpoonright \mathbf{p}$ where $\mathbf{p} \in role(\mathbf{G})$, which implies that $obj(C)$, typed by I, plays as $\mathbf{p}$ in $\mathbf{G}$ when it joins $\mathbf{s}$. Then we extend Θ to Θ' by adding $I : \mathbf{p}$ into $\{\varphi\}_{\mathbf{s}:\mathbf{G}}$ to claim that $\mathbf{p}$ associates to interface I, and replacing $\mathbf{s}[obj(C)] : \emptyset$ with $\mathbf{s}[\mathbf{p}] : \mathbf{L}$ in the channel environment.

Rule (T-Send) types an asynchronous remote method call. The object is allowed to have such a call, specified by $\mathbf{s}[\mathbf{p}] : \mathbf{q}!_f m$, when the object calls a method m using f (by checking $x : f$) at an object playing $\mathbf{q}$ in $\mathbf{G}$ (i.e. Θ has $\{\varphi, I : \mathbf{q}\}_{\mathbf{s}:\mathbf{G}}$ and Γ has $e : I$) and its next statement s is also well-typed.

Rule (T-Method) types a method execution. It is valid to do so if the method body is well-typed and the caller is an object playing $\mathbf{q}$ in $\mathbf{G}$, known by $\Theta = \Theta', \{\varphi, T : \mathbf{q}\}_{\mathbf{s}:\mathbf{G}}$ where T is either equal to I or Any. We use f' for returning the computation result as long as $f' = f$ (i.e. $pjm(\mathbf{q}?_f m.\mathbf{L}, f') = \mathbf{q}?_f m.pjm(\mathbf{L}, f')$.)

Rule (T-Offer) types $\mathtt{case}\ e\{p_1 \Rightarrow s_1, ..., p_n \Rightarrow s_n\}$ with $\&_f\{\mathbf{L}_j\}_{j\in J}$ by checking if every branch $p_j \Rightarrow s_j$ is well-typed by $\mathbf{L}_j$ and checking if f is active in $\mathbf{L}_j$ on the object $\mathbf{p}$ in session $\mathbf{s}$. Rule (T-Choice) is the counterpart of (T-Offer). Rule (T-Await) types the await statement with $\mathsf{Await}(f, f')$. It checks if the next statament is well-typed by $\mathsf{React}(f).\mathbf{L}'$, which specifies the next action is to reactivate the usage of f and, since f' has been resolved, in $\mathbf{L}'$ we have f' inactive on $\mathbf{p}$. Other rules are straightforward.

After locally type checking every objects' implementations in a session based on their corresponding local types, which are projected from a global type that the session follows, our system ensures overall interactions among those objects comply with communication correctness:

Theorem 1 (Communication Correctness). *Let $\boldsymbol{G}$ be a projectable global type and S a closed system in which a session $\boldsymbol{s}$ obeys to $\boldsymbol{G}$. Let $\mathbf{p'_1}, ..., \mathbf{p'_n}$ are objects in $\boldsymbol{s}$ and respectively act as $\mathbf{p_1}, ..., \mathbf{p_n}$ in $\boldsymbol{G}$. If the objects' implementations are all well-typed by rules in Fig. 5, the interactions among $\mathbf{p'_1}, ..., \mathbf{p'_n}$ comply with communication correctness against $\boldsymbol{G}$.*

6 Session Automata

As we type check a SABS program via rules in Fig.5, by rule (T-Scheduler), a local type $\mathbf{L}$ is assigned as a *scheduling policy* to the scheduler of object

[Scheduler: L] `class` $C\{...\}$; the scheduler can (re)activate processes (i.e. by executing methods) based on **L**. To ensure that the scheduler's behavior follows **L**, we propose a verification mechanism where a scheduler uses a session automaton [2], as a *scheduling policy*, to model the possible sequence of events.

In this model, when the object is idle, the object's scheduler inputs the processes which can be (re)activated according to the session automaton, which is automatically generated by **L**. If a labelled transition, which corresponds to an event, can fire in the automaton, the object (re)activates this event. If there are several processes which can run such transition, the scheduler randomly selects one of them. This mechanism is a variant of typestate [20].

Session automata, a subclass of register automata [17], only store fresh futures; it is decideable whether two session automata accept the same language [2]:

Definition 3. [k-Register Session Automata] Let Σ be a finite set of labels, D be an infinite set of data equipped with equality, and $k \in \mathbb{N}$. A *k-Register Session Automaton* is a tuple (Q, q_0, Φ, F) where Q is the finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\Phi \subseteq (Q \times Q') \cup (Q \times \Sigma \times 2^{\{1,\ldots,k\}} \times \{1,\ldots,k\} \times Q)$ is the transition relation.

Data words are words over an alphabet $\Sigma \times D$. A data word automata has a data store, which can save k data values. A transition fires for a letter $(a, d) \in \Sigma \times D$ if a set of equalities of the form $d = r_i$ are satisfied, where r_i refers to the ith stored data value. After a transition fires, the data store records d.

Let $\sigma : \{1,\ldots,k\} \to D$ be the store. We define (q, a, I, i, q') as a transition in automaton from state q to state q' upon reading (a, d) if $\sigma[i] = d$ for all $i \in I$, $I = \{1..n\}$ by updating $\sigma[i]$ to d, and define (q, q') as an ϵ-transition that switches the state without reading the next letter:

Definition 4. [Runs of Session Automata] A *run* of a k-register session automaton $A = (Q, q_0, \Phi, F)$ on a data word $w = (a_1, d_1), \ldots, (a_k, d_k) \in (\Sigma \times D)^k$ is a sequence $s \in \big(Q \times \mathbb{N} \times (\{1,\ldots,k\} \to D)\big)^*$. An element (q, j, σ) of the sequence denotes that A is at state q with store σ and reads (a_j, d_j). To be a run of A, the sequence $s = (q_0, j_0, \sigma_0), \ldots, (q_n, j_n, \sigma_n)$ must satisfy the following:

$$\Big((q_i, q_{i+1}) \in \Phi \wedge (j_i = j_{i+1}) \wedge (\sigma_i = \sigma_{i+1})\Big) \vee$$
$$\Big((q_i, (a_{j_i}, d_{j_i}), I, k, q_{i+1}) \in \Phi \wedge (j_{i+1} = j_i + 1) \wedge (\sigma_{i+1} = \sigma_i[k/d_{j_i}]) \wedge \forall l \in I.\ \sigma_i(l) = d_{j_i}\Big)$$

where $I = \{1..n\}$ and $\sigma_i[k/d_{j_i}]$ is a function mapping the kth stored data to d_{j_i}. Now we revise Σ to $\Sigma = \big((\{\texttt{invocREv}\} \times \mathsf{Met}) \cup \{\texttt{reactEv}\}\big)$ and $D = \mathsf{Fut}$, where `invocREv` labels process activation and `reactEv` labels process reactivation. Given an object type, we can build a session automaton:

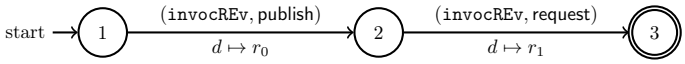
Definition 5. Let **L** be an object type. Let k be number of futures in $\hat{\mathbf{L}}$. We assume the futures are ordered and $pos(f)$ refers to the number of f in the ordering. The k-register session automaton $A_{\mathbf{L}}$ is defined inductively as follows:

- $\mathbf{p}?_f m$ is mapped to a 2-state automaton which reads $(\texttt{invocREv}, m)$ and stores the future f in the $pos(f)$-th register on its sole transition.

– React f is mapped to a 2-state automaton which reads `reactEV` and tests for equality with the $pos(f)$-th register on its sole transition.
– Concatenation, branching, and repetition using the standard construction for concatenation, union, and repetition for NFAs.

When a process is activated, the automaton stores the process's corresponding futures; when a process is reactivated, the automaton compares the process's corresponding futures with the specified register. As all repetitions in types projected from a global type are self-contained, after the repetition, the futures used there are resolved and thus the automaton can overwrite it safely. The example below shows how a session automaton works based on an object type:

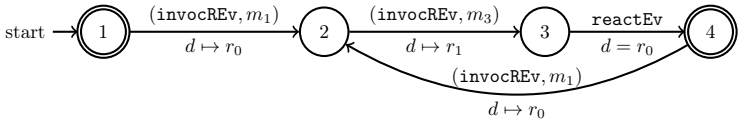
Example 4. Consider the example from Section 2. A simple automaton describing the sequence for the **d** (Service Desk) is



The scheduler above does not need to use registers because it does not have reactivations. The following one must read the registers to schedule reactivations:

$$\mathbf{L} = (\mathbf{p}?_f m_1.\mathsf{Await}(f, f').\mathbf{p}?_{f''} m_3.\mathsf{Put}\ f''.\mathsf{React}(f).\mathsf{Put}\ f)^*$$

The generated 2-register session automaton is:



The following theorem states that the objects involving in a session are faithful to the session's protocol if their processes can be verified by the corresponding schedulers, whose behaviors follow the session automata:

Theorem 2 (Fidelity). *Let $\boldsymbol{G}$ be a projectable global type and S a closed system in which a session $\boldsymbol{s}$ obeys to $\boldsymbol{G}$. Let $\mathbf{p'_1}, ..., \mathbf{p'_n}$ are objects interacting in $\boldsymbol{s}$ and respectively act as participants $\mathbf{p_1}, ..., \mathbf{p_n}$ in $\boldsymbol{G}$. Let A_j be a session automaton generated from $\boldsymbol{G} \upharpoonright \mathbf{p_j}$. If every scheduler for $\mathbf{p_j}$, $j \in \{1..n\}$ accepts the same language as A_j does, the implementations on objects are faithful to $\boldsymbol{G}$.*

7 Related and Future Work

The compositional approach introduced in [5, 6] proposed a four event semantics for core ABS. Their verification approach was bottom-up, i.e., class invariants are verified and composed into system property based on history well-formedness; while our approach is top-down, i.e., system property is specified in session types and projected into class invariants. We verify class invariants based on the scheduling policy type-checked by local session types. Besides, we introduce session types for process suspension and process reactivation.

Session types for object-oriented languages have been studied in [3, 8] and implemented for libraries/extensions of mainstream languages like Java [13].

Also, lightweight session programming in Scala [19] was proposed by introducing a representation of session types as Scala types. However, they do not explore the valid usage of futures for modeling channel-based concurrency, neither verify cooperative scheduling against specified execution orders.

Schedulers with automata for actor-based models were studied by Jaghoori et al. [14], while user-defined schedulers for ABS were introduced by Bjørk et al. [1]. Our use of automata is similar to the *drivers* of [14], where drivers can not reject any process if the process queue is non-empty and do not consider reactivations. Our schedulers enable the object to wait for a method call to arrive.

Field et al. [7] used finite state automata (without registers) to encode typestate, Gay et al. [8] used typestate to guide session types with non-uniform objects, while Grigore et al. [9] established register automata for runtime verification. In their approach the automata monitor the order of method invocations in a sequential setting. The registers are used to store an unbounded amount of object identities. Our automata are extended to be able to check the specified orders of method calls. The schedulers thus can apply these automta to schedule specified activations and reactivations in a concurrent setting.

Neykova and Yoshida [18] also consider an actor model with channels, where processes are monitored by automata. Deniélou and Yoshida [4] used communicating automata to approximate processes and local types. However, their approaches do not consider scheduling and validating the usage of futures.

We plan to prove that our type system ensures that interactions among objects are deadlock-free and always progresses, and then implement a session-based extension for the core ABS language.

8 Conclusion

We establish a hybrid framework for compositional analysis. The system property is guaranteed by type checking each objects' behaviors against local session types, which are gained by projecting global types on endpoints. In summary, we statically ensure communication correctness for concurrent processing and, at the same time, ensure local schedulers' behaviors will follow the specified execution order among asynchronous communications at runtime.

9 Acknowledgments

We thank Reiner Hähnle and Patrick Eugster who provided the original idea and insightful discussions for this paper. We also thank the reviewers for their constructive comments. This work was supported by the ERC grant FP7-617805 *LiVeSoft: Lightweight Verification of Software* and the EU project FP7-610582 *Envisage: Engineering Virtualized Services.*

References

1. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.
2. B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In M. Béal and O. Carton, editors, *DLT'13*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

3. J. Campos and V. T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In K. Honda and A. Mycroft, editors, *PLACES'10*, volume 69 of *EPTCS*, pages 12–28, 2010.
4. P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In H. Seidl, editor, *ESOP'12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
5. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *CADE'15*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015.
6. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
7. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.
8. S. J. Gay, N. Gesbert, A. Ravara, and V. T. Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.
9. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In N. Piterman and S. A. Smolka, editors, *TACAS'13*, volume 7795 of *LNCS*, pages 260–276. Springer, 2013.
10. R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO'12*, volume 7866 of *LNCS*, pages 1–37. Springer, 2012.
11. R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1985.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL'08*, pages 273–284. ACM, 2008.
13. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In J. Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
14. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *The Journal of Logic and Algebraic Programming*, 78(5):402 – 416, 2009.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2010.
16. E. Kamburjan. Session Types for ABS. Technical report, 2016. www.se.tu-darmstadt.de/publications/details/?tx_bibtex_pi1[pub_id]=tud-cs-2016-0179.
17. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
18. R. Neykova and N. Yoshida. Multiparty session actors. In E. Kühn and R. Pugliese, editors, *COORDINATION'14 Proceedings*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.
19. A. Scalas and N. Yoshida. Lightweight session programming in Scala. In S. Krishnamurthi and B. S. Lerner, editors, *ECOOP'16*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
20. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.
21. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

Appendix B

Combining Static Analysis and Testing for Deadlock Detection

Combining Static Analysis and Testing for Deadlock Detection *

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel

Complutense University of Madrid (UCM), Spain

Abstract. Static deadlock analyzers might be able to verify the absence of deadlock. However, they are usually not able to detect its presence. Also, when they detect a potential deadlock cycle, they provide little (or even no) information on their output. Due to the complex flow of concurrent programs, the user might not be able to find the source of the anomalous behaviour from the abstract information computed by static analysis. This paper proposes the combined use of static analysis and testing for effective deadlock detection in asynchronous programs. When the program features a deadlock, our combined use of analysis and testing provides an effective technique to catch deadlock traces. While if the program does not have deadlock, but the analyzer inaccurately spotted it, we might prove deadlock freedom.

1 Introduction

In concurrent programs, *deadlocks* are one of the most common programming errors and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. We consider an *asynchronous* language which allows spawning asynchronous tasks at distributed locations, with no shared memory among them, and which has two operations for blocking and non-blocking synchronization with the termination of asynchronous tasks. In this setting, in order to detect deadlocks, all possible *interleavings* among tasks executing at the distributed locations must be considered. Basically, each time that the processor can be released, any of the available tasks can start its execution, and all combinations among the tasks must be tried, as any of them might lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks. As static analysis examines all possible execution paths and variable values, it can reveal deadlocks that could not manifest until weeks or months after releasing the application. This aspect of static analysis is especially important in security assurance – security attacks try to exercise an application in unpredictable and untested ways. However, due to the use of approximations, most static analyses can only verify the absence of deadlock but not its presence, i.e., they can produce false positives. Moreover, when a deadlock is found, state-of-the-art analysis tools

* This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

[6, 7, 12] provide little (and often no) information on the source of the deadlock. In particular, for deadlocks that are complex (involve many tasks and locations), it is essential to know the task interleavings that have occurred and the locations involved in the deadlock, i.e., provide a concrete *deadlock trace* that allows the programmer to identify and fix the problem.

In contrast, testing consists of executing the application for concrete input values. Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings. The primary advantage of *systematic testing* [4, 14] for deadlock detection is that it can provide the detailed deadlock trace. There are two shortcomings though: (1) Although recent research tries to avoid redundant exploration as much as possible [1, 3–5], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming. (2) There is only guarantee of deadlock freedom for finite-state terminating programs (terminating executions with concrete inputs).

This paper proposes a seamless combination of static analysis and testing for effective deadlock detection as follows: an existing static deadlock analysis [6] is first used to obtain *abstract* descriptions of potential deadlock cycles which are then used to guide a testing tool in order to find associated deadlock traces (or discard them). In summary, the main contributions of this paper are:

1. We extend a standard semantics for asynchronous programs with information about the task interleavings made and the status of tasks.
2. We provide a formal characterization of *deadlock state* which can be checked along the execution and allows us to early detect deadlocks.
3. We present a new methodology to detect deadlocks which combines testing and static analysis as follows: the deadlock cycles inferred by static analysis are used to guide the testing process towards paths that might lead to a deadlock cycle while discarding deadlock-free paths.
4. We have implemented our methodology in the SYCO system (see Sect. 6) and performed a thorough experimental evaluation on some classical examples.

2 Asynchronous Programs: Syntax and Semantics

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffers of any processor, including its own, and synchronize with the termination of tasks. The language uses *future variables* to check if the execution of an asynchronous task has finished. An asynchronous call $\mathsf{m}(\bar{\mathsf{z}})$ spawned at location x is associated with a future variable f as follows $\mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}})$. Instructions $f.\mathsf{block}$ and $f.\mathsf{await}$ allow, respectively, blocking and non-blocking synchronization with the termination of m. When a task completes, or when it is awaiting with a non-blocking await

$$\text{(MSTEP)}\ \frac{selectLoc(S) = loc(\ell, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, selectTask(\ell) = tsk(tk, m, l, s), \quad S \diamond \boldsymbol{\rho_\emptyset} \overset{\ell \cdot tk}{\leadsto}{}^{*}\ S' \diamond \boldsymbol{\rho}}{S \xrightarrow{\ell \cdot tk} S'}$$

$$\text{(NEWLOC)}\ \frac{tk = tsk(tk, m, l, pp{:}x = \mathsf{new}\ D; s), \mathrm{fresh}(\ell'), h' = newheap(D), l' = l[x \to \ell']}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \rightsquigarrow loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l', s)\}) \cdot loc(\ell', \bot, h', \{\}) \diamond \boldsymbol{\rho_0}}$$

$$\text{(ASYNC)}\ \frac{tk = tsk(tk, m, l, pp{:}y{=}x!m_1(\bar{z}); s), l(x){=}\ell_1,\ \mathrm{fresh}(tk_1),\ l_1{=}buildLocals(\bar{z}, m_1, l)}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot loc(\ell_1, \_, \_, \mathcal{Q}') \diamond \boldsymbol{\rho_0} \rightsquigarrow loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot \\ loc(\ell_1, \_, \_, \mathcal{Q}' \cup \{tsk(tk_1, m_1, l_1, body(m_1))\}) \cdot fut(y, o_1, tk_1, ini(m_1)) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$\text{(RETURN)}\ \frac{tk = tsk(tk, m, l, pp{:}\mathsf{return}; s), \boldsymbol{\rho_1} = \mathsf{return}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \diamond \boldsymbol{\rho_0} \rightsquigarrow loc(\ell, \bot, h, \mathcal{Q} \cup \{tsk(tk, m, l, \epsilon)\}) \diamond \boldsymbol{\rho_1}}$$

$$\text{(AWAIT1)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \rightsquigarrow \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$\text{(AWAIT2)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{await}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{await}}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \rightsquigarrow \\ loc(\ell, \bot, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1} \end{array}}$$

$$\text{(BLOCK1)}\ \frac{tk = tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 = \epsilon}{\begin{array}{c} loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \rightsquigarrow \\ loc(\ell, tk, h, \mathcal{Q} \cup \{tsk(tk, m, l, s)\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \end{array}}$$

$$\text{(BLOCK2)}\ \frac{tk{=}tsk(tk, m, l, pp{:}y.\mathsf{block}; s), tsk(tk_1, \_, \_, s_1) \in \mathsf{Loc}, s_1 \neq \epsilon, \boldsymbol{\rho_1} = pp{:}y.\mathsf{block}}{loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_0} \rightsquigarrow loc(\ell, tk, h, \mathcal{Q} \cup \{tk\}) \cdot fut(y, \_, tk_1, \_) \diamond \boldsymbol{\rho_1}}$$

Fig. 1. Macro-Step Semantics of Asynchronous Programs

for a task that has not finished yet, its processor becomes idle again, chooses the next pending task, and so on. The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to a *concurrent object* and can be dynamically created using the instruction **new**. The program consists of a set of methods of the form $M ::= T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form $s ::= s; s \mid x{=}e \mid$ **if** e then s **else** $s \mid$ **while** e **do** $s \mid$ **return** $\mid b{=}$**new** $\mid f = x\ !\ m(\bar{z}) \mid f.\mathsf{await} \mid f.\mathsf{block}$. For the sake of generality, the syntax of expressions e and types T is left open.

Fig. 1 presents the semantics of the language. The information about $\boldsymbol{\rho}$ in bold font is part of the extensions for testing in Sec. 4 and should be ignored for now. A *state* or *configuration* is a set of locations and future variables $loc_0 \cdots loc_n \cdot fut_0 \cdots fut_m$. A *location* is a term $loc(\ell, tk, h, \mathcal{Q})$ where ℓ is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\bot$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *future variable* is a term $fut(id, \ell, tk, m)$ where id is a unique future variable identifier, ℓ is the location identifier that executes the task tk awaiting for the future, and m is the initial program point of tk. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated. We

assume that the execution starts from a main method without parameters. The initial state is St=$\{loc(0, 0, \bot, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0. Here, l maps local variables to their initial values (**null** in case of reference variables) and $\bot$ is the empty heap. $body(m)$ is the sequence of instructions in method m, and we can know the program point pp where an instruction s is in the program as follows pp:s.

As locations do not share their states, the semantics can be presented as a macro-step semantics [14] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to an await or return instruction. In this case, we apply rule MSTEP to select an available task from a location, namely we apply the function $selectLoc(S)$ to select non-deterministically one *active* location in the state (i.e., a location with a non-empty queue) and $selectTask(\ell)$ to select non-deterministically one task of ℓ's queue. The transition $\rightsquigarrow$ defines the evaluation within a given location. NEWLOC creates a new location without tasks, with a fresh identifier and heap. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tk_1, and it adds a new future to the state. $ini(m)$ refers to the first program point of method m. We assume $\ell \neq \ell_1$, but the case $\ell = \ell_1$ is analogous, the new task tk_1 is added to $\mathcal{Q}$ of ℓ. The rules for sequential execution are standard and are thus omitted. AWAIT1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task t_1 is only looked up but it does not disappear from the state as its status may be needed later on. AWAIT2: Otherwise, the task yields the lock so that any other task of the same location can take it. RETURN: When **return** is executed, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction ϵ). BLOCK2: A y.block instruction waits for the future variable but without yielding the lock. Then, when the future is ready, BLOCK1 allows continuing the execution.

In what follows, a *derivation* or *execution* $E \equiv St_0 \longrightarrow \cdots \longrightarrow St_n$ is a sequence of macro-steps (applications of rule MSTEP). The derivation is *complete* if St_0 is the initial state and $\nexists\ St_{n+1} \neq St_n$ such that $St_n \longrightarrow St_{n+1}$. Since the execution is non-deterministic, multiple derivations are possible from a state. Given a state St, $exec(St)$ denotes the set of all possible derivations starting at St. We sometimes label transitions with $\ell \cdot tk$, the name of the location ℓ and task tk selected (in rule MSTEP) or evaluated in the step (in the transition $\rightsquigarrow$). The systematic exploration of $exec(St)$ thus corresponds to the standard systematic testing setting with no reduction of any kind.

3 Motivating Example

Our running example is a simple version of the classical sleeping barber problem where a barber sleeps until a client arrives and takes a chair, and the client wakes up the barber to get a haircut. Our implementation in Fig. 2 has a main method shown on the left and three classes Ba, Ch and Cl implementing the barber, chair and client, respectively. The main creates three locations barber, client and chair and spawns two asynchronous tasks to start the wakeup task in the client and

```
1 main() {
2  Ba barber = new Ba();
3  Cl client = new Cl();
4  Ch chair = new Ch();
5  client!wakeup(barber,chair);
6  barber!sleeps(client,chair);
7 }
8 class Ba{
9   Unit sleeps(Cl cl, Ch ch){
10     Fut f=ch!taken(cl);
11     f.block;}
12   Unit cuts(){}
13 }
```

```
14 class Ch{
15   Unit taken(Cl cl){
16     Fut f=cl!sits();
17     f.await;}
18   Unit isClean(){}
19 }
20 class Cl{
21   Unit wakeup(Ba b, Ch ch){
22     Fut f=b!cuts();
23     ch!isClean();
24     f.block;}
25   Unit sits(){}
26 }
```

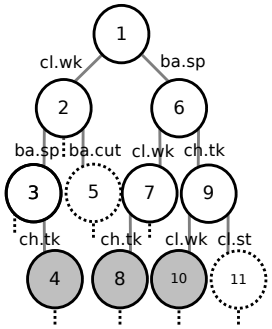


Fig. 2. Classical Sleeping Barber Problem (left) and Execution Tree (right)

sleeps in the barber, both tasks can run in parallel. The execution of sleeps spawns an asynchronous task on the chair to represent the fact that the client takes the chair, and then blocks at line 11 (L11 for short) until the chair is taken. The task taken first adds the task sits on the client, and then awaits on its termination at L17 without blocking, so that another task on the location chair can execute. On the other hand, the execution of wakeup in the client spawns an asynchronous task cuts on the barber and one on the chair, isClean, to check if the chair is clean. The execution of the client blocks until cuts has finished. We assume that all methods have an implicit return at the end.

Fig. 2 summarizes the systematic testing tree of the main method by showing some of the macro-steps taken. Derivations that contain a dotted node are not deadlock, while those with a gray node are deadlock. A main motivation of our work is to detect as early as possible that the dotted derivations will not lead us to deadlock and prune them. Let us see two selected derivations in detail. In the derivation ending at node 5, the first macro-step executes cl.wakeup and then ba.cuts. Now, it is clear that the location cl will not deadlock, since the block at L24 will succeed and the other two locations will be also able to complete their tasks, namely the await at L17 of location ch can finish because the client is certainly not blocked, and also the block at L11 will succeed because the task in taken will eventually finish as its location is not blocked. However, in the branch of node 4, we first select wakeup (and block client), then we select sleeps (and block barber), and then select taken that will remain in the await at L17 and will never succeed since it is awaiting for the termination of a task of a blocked location. Thus, we have a deadlock. Let us outline five states of this derivation:

$St_1 \equiv loc(ini,..)\cdot loc(cl,..,\{tsk(1,wk,..)\})\cdot loc(ba,..,\{tsk(2,sp,..)\})\cdot loc(ch,..) \xrightarrow{cl,1}$
$St_2 \equiv loc(cl,..,\{tsk(1,wk,f_0.block)\})\cdot loc(ba,..,\{tsk(3,cut,..),..\})\cdot fut(f_0,ba,3,12)\cdot.. \xrightarrow{ba,2}$
$St_3 \equiv loc(ba,..,\{tsk(2,sp,f_1.block)\})\cdot loc(ch,..,\{tsk(5,tk,..),..\})\cdot fut(f_1,ch,5,15)\cdot.. \xrightarrow{ch,5}$
$St_4 \equiv loc(ch,..,\{tsk(5,tk,f_2.await),..\})\cdot loc(cl,..,\{tsk(6,st,..),..\})\cdot fut(f_2,cl,6,25)\cdot..$
$\xrightarrow{ch,4} St'_4 \equiv loc(ch,..\{tsk(4,isClean,\epsilon),..\})\cdot..$

$$\text{(MSTEP2)}\quad \frac{\begin{array}{c} selectLoc(S) = loc(\ell, \bot, h, \mathcal{Q}),\ \mathcal{Q} \neq \emptyset,\ selectTask(\ell) = tsk(tk, m, l, pp : s), \\ \textbf{check}_{\mathfrak{C}}(\boldsymbol{S, table}),\ S \diamond \rho_0 \overset{\ell \cdot tk}{\rightsquigarrow^*} S' \diamond \rho,\ S \neq S',\ \boldsymbol{not}(\mathsf{deadlock}(\boldsymbol{S'})) \\ clock(n),\ table' = table \cup t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle \end{array}}{(S, table) \xrightarrow{\ell \cdot tk} (S', table')}$$

Fig. 3. MSTEP2 rule for combined testing and analysis

The first state is obtained after executing the main where we have the initial location *ini*, three locations created at L2, L3 and L4, and two tasks at L5 and L6 added to the queues. Note that each location and task is assigned a unique identifier (we use numbers as identifiers for tasks and short names as identifiers for locations). In the next state, the task wakeup has been selected and fully executed (we have shortened the name of the methods, e.g., wk for wakeup). Observe at St_2 the addition of the future variable created at L22. In St_3 we have executed task sleeps in the barber and added a new future term. In St_4 we execute task taken in the chair (this state is already deadlock as we will see in Sec. 4.2), however location chair can keep on executing an available task isClean generating St'_4. From now on, we use the location and task names instead of numeric identifiers for clarity.

4 Testing for Deadlock Detection

The goal of this section is to present a framework for early detection of deadlocks during systematic testing. This is done by enhancing our standard semantics with information which allows us to easily detect *dependencies* among tasks, i.e., when a task is awaiting for the termination of another one. These dependencies are necessary to detect in a second step *deadlock states*.

4.1 An Enhanced Semantics for Deadlock Detection

In the following we define the *interleavings table* whose role is twofold: (1) It stores all decisions about task interleavings made during the execution. This way, at the end of a concrete execution, the exact ordering of the performed macro-steps can be observed. (2) It will be used to detect deadlocks as early as possible, and, also to detect states from which a deadlock cannot occur, therefore allowing to prune the execution tree when we are looking for deadlocks. The interleavings table is a mapping with entries of the form $t_{\ell,tk,pp} \mapsto \langle n, \rho \rangle$, where:

- $t_{\ell,tk,pp}$ is a *macro-step identifier*, or *time identifier*, that includes: the identifiers of the location ℓ and task tk that have been selected in the macro-step, and the program point pp of the first instruction that will be executed;
- n is an integer representing the time when the macro-step starts executing;
- ρ is the status of the task after the macro-step and it can take three values as it can be seen in Fig. 1: block or await when executing these instructions on a future variable that is not ready (we also annotate in ρ the information on the associated future); **return** that allows us to know that the task finished.

We use a function ***clock*(n)** to represent a clock that starts at 0, is increased by one in every execution of ***clock***, and returns the current value **n**. The initial entry is $t_{0,0,1} \mapsto \langle 0, \rho_0 \rangle$, 0 being the identifier for the initial location and task,

and 1 the first program point of *main*. The clock also assigns the value 0 as the first element in the tuple and a fresh variable in the the second element ρ_0. The next macro-step will be assigned clock value 1, next 2, and so on. As notation, we define the relation $t \in table$ if there exists an entry $t \mapsto \langle n, \rho\rangle \in table$, and the function $status(t, table)$ which returns the status ρ_t such that $t \mapsto \langle n, \rho_t\rangle \in table$. The semantics is extended by changing rule MSTEP as in Fig. 3. The function deadlock will be defined in Thm. 1 to stop derivations as soon as deadlock is detected. Function $\mathsf{check}_{\mathfrak{C}}$ should be ignored for now, it will be defined in Sec. 5.2. Essentially, there are two new aspects: (1) The state is extended with the status ρ, namely all rules include a status ρ attached to the state using the symbol $\diamond$. The status is showed in bold font in Fig. 1 and can get a value in rules block2, await2 and return. The initial value ρ_0 is a fresh variable. (2) The state for the macrostep is extended with the interleavings table *table*, and a new entry $t_{\ell,tk,pp} \mapsto \langle n, \rho\rangle$ is added to *table* in every macrostep if there has been progress in the execution, i.e., $S' \neq S$, n being the current clock time.

Example 1. The interleavings table below (left) is computed for the derivation in Sec. 3. It has as many entries as macro-steps in the derivation. We can observe that subsequent time values are assigned to each time identifier so that we can then know the order of execution. The right column shows the future variables in the state that store the location and task they are bound to.

St_1	$t_{ini,main,1} \mapsto \langle 0, return\rangle$	$\emptyset$
St_2	$t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block\rangle$	$fut(f_0, ba, cuts, 12)$
St_3	$t_{ba,sleeps,9} \mapsto \langle 2, 11{:}f_1.block\rangle$	$fut(f_1, ch, taken, 15)$
St_4	$t_{ch,taken,15} \mapsto \langle 3, 17{:}f_2.await\rangle$	$fut(f_2, cl, sits, 25)$

4.2 Formal Characterization of Deadlock State

Our semantics can easily be extended to detect deadlock just by redefining function *selectLoc* so that only locations that can proceed are selected. If, at a given state, no location is selected but there is at least a location with a non-empty queue then there is a deadlock. However, deadlocks can be detected earlier. We present the notion of *deadlock state* which characterizes states that contain a *deadlock chain* in which one or more tasks are waiting for each other's termination and none of them can make any progress. Note that, from a deadlock state, there might be tasks that keep on progressing until the deadlock is finally made explicit. Even more, if one of those tasks runs into an infinite loop, the deadlock will not be captured using this naive extension. The early detection of deadlocks is crucial to reduce state exploration as our experiments show in Sec. 6.

We first introduce the auxiliary notion of *waiting interval* which captures the period in which a task is waiting for another one to terminate. In particular, it is defined as a tuple $(t_{stop}, t_{async}, t_{resume})$ where t_{stop} is the macro-step at which the location stops executing a task due to some block/await instruction, t_{async} is the macro-step at which the task that is being awaited is selected for execution, and, t_{resume} is the macro-step at which the task will resume its execution. t_{stop}, t_{async} and t_{resume} are time identifiers as defined in Sec. 4.1. t_{resume} will also be written as $next(t_{stop})$. When the task stops at t_{stop} due to a block instruction,

we call it *blocking interval*, as the location remains blocked between t_{stop} and $next(t_{stop})$ until the awaited task, selected in t_{async}, has already finished. The execution of a task can have several points at which macro-steps are performed (e.g., if it contains several await or block the processor may be lost several times). For this reason, we define the set of successor macro-steps of the same task from a macro-step: $suc(t_{\ell,tk,pp_0}, table) = \{t_{\ell,tk,pp_i} : t_{\ell,tk,pp_i} \in table, t_{\ell,tk,pp_i} \geq t_{\ell,tk,pp_0}\}$.

Definition 1 (Waiting/Blocking Intervals). *Let $St = (S, table)$ be a state, $I = (t_{stop}, t_{async}, t_{resume})$ is a waiting interval of St, written as $I \in St$, iff:*
1. $\exists\, t_{stop} = t_{\ell,tk_0,pp_0} \in table,\ \rho_{stop} = status(t_{stop}) \in \{pp_1 : x.\mathsf{await}, pp_1{:}x.\mathsf{block}\}$,
2. $t_{resume} \equiv t_{\ell,tk_0,pp_1},\ fut(x, \ell_x, tk_x, pp(M)) \in S$,
3. $t_{async} \equiv t_{\ell_x,tk_x,pp(M)}, \nexists\, t \in suc(t_{async}, table)$ with $status(t) = return$.
If $\rho_{stop} = x.\mathsf{block}$, then I is blocking.

In condition 3, we can see that if the task starting at t_{async} has finished, then it is not a waiting interval. This is known by checking that this task has not reached return, i.e., $\nexists\ t \in suc(t_{async}, table)$ such that $status(t) = return$. In condition 1, we see that in ρ_{stop} we have the name of the future we are awaiting (whose corresponding information is stored in fut, condition 2). In order to define t_{resume} in condition 2, we search for the same task tk_0 and same location ℓ that executes the task starting at program point pp_1 of the await/block, since this is the point that the macro-step rule uses to define the macro-step identifier t_{ℓ,tk_0,pp_1} associated to the resumption of the waiting task.

Example 2. Let us consider again the derivation in Sec. 3. We have the following blocking interval $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_2$ with $St_2 \equiv (S_2, table_2)$, since $t_{cl,wakeup,21} \in table_2$, $status(t_{cl,wakeup,21}, table_2) = [24{:}f.\mathsf{block}]$, $(f, ba, cuts, 12) \in St_2$ and $t_{ba,cuts,12} \notin table_2$. This blocking interval captures the fact that the task at $t_{cl,wakeup,21}$ is blocked waiting for task *cuts* to terminate. Similarly, we have the following two intervals in St_4: $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ and $(t_{ch,taken,15}, t_{cl,sits,25}, t_{ch,taken,17})$.

The following notion of *deadlock chain* relies on the waiting/blocking intervals of Def. 1 in order to characterize chains of calls in which intuitively each task is waiting for the next one to terminate until the last one which is waiting on the termination of a task executing on the initial location (that is blocked). Given a time identifier t, we use $loc(t)$ to obtain its associated location identifier.

Definition 2 (Deadlock Chain). *Let $St = (S, table)$ be a state. A chain of time identifiers $t_0, ..., t_n$ is a deadlock chain in St, written as $dc(t_0, ..., t_n)$ iff $\forall t_i \in \{t_0, ..., t_{n-1}\}$ s.t. $(t_i, t'_{i+1}, next(t_i)) \in St$ one of the following conditions holds:*
1. $t_{i+1} \in suc(t'_{i+1}, table)$, or
2. $loc(t'_{i+1}) = loc(t_{i+1})$ and $(t_{i+1}, \_, next(t_{i+1}))$ is blocking.
and for t_n, we have that $t_{n+1} \equiv t_0$, and condition 2 holds.

Let us explain the two conditions in the above definition: In condition (1), we check that when a task t_i is waiting for another task to terminate, the waiting interval contains the initial time t'_{i+1} in which the task will be selected. However, we look for any waiting interval for this task t_{i+1} (thus we check that t_{i+1} is a

successor of time t'_{i+1}). As in Def. 2, this is because such task may have started its execution and then suspended due to a subsequent await/block instruction. Abusing terminology, we use the time identifier to refer to the task executing. In condition (2), we capture deadlock chains which occur when a task t_i is waiting on the termination of another task t'_{i+1} which executes on a location $loc(t'_{i+1})$ which is blocked. The fact that is blocked is captured by checking that there is a blocking interval from a task t_{i+1} executing on this location. Finally, note the circularity of the chain, since we require that $t_{n+1} \equiv t_0$.

Theorem 1 (Deadlock state). *A state St is deadlock, written* $\mathsf{deadlock}(S)$*, if and only if there is a deadlock chain in St.*

Derivations ending in a deadlock state are considered complete derivations. We prove that our definition of deadlock is equivalent to the standard definition of deadlock in [6] (proof can be found in [16]).

Example 3. Following Ex. 1, St_4 is a deadlock state since there exists a *deadlock chain* $dc(t_{cl,wakeup,21}, t_{ba,sleeps,9}, t_{ch,taken,15})$. For the second element in the chain $t_{ba,sleeps,9}$, condition 1 holds as $(t_{ba,sleeps,9}, t_{ch,taken,15},\ t_{ba,sleeps,11}) \in St_4$ and $t_{ch,taken,15} \in suc(t_{ch,taken,15}, table_4)$. For the first element $t_{cl,wakeup,21}$, condition 2 holds since $(t_{cl,wakeup,21}, t_{ba,cuts,12}, t_{cl,wakeup,24}) \in St_4$ and $(t_{ba,sleeps,9}, t_{ch,taken,15}, t_{ba,sleeps,11})$ is blocking. Condition 2 holds analogously for $t_{ch,taken,15}$.

5 Combining Static Deadlock Analysis and Testing

This section proposes a deadlock detection methodology that combines static analysis and systematic testing as follows. First, a state-of-the-art deadlock analysis is run, in particular that of [6], which provides a set of abstractions of potential *deadlock cycles*. If the set is empty, then the program is deadlock-free. Otherwise, using the inferred set of deadlock cycles, we systematically test the program using a novel technique to guide the exploration towards paths that might lead to deadlock cycles. The goals of this process are: (1) finding concrete deadlock traces associated to the feasible cycles, and, (2) discarding unfeasible deadlock cycles, and in case all cycles are discarded, ensure deadlock freedom for the considered input or, in our case, for the main method under test. As our experiments show in Section 6, our technique allows reducing significantly the search space compared to the full systematic exploration.

5.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [6] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. Three kinds of arrows can be distinguished, namely, *task-task* (a task is awaiting for the termination of another one), *task-location* (a task is awaiting for a location to be idle) and *location-task* (the location is blocked due the task). *Location-location* arrows cannot happen. The abstractions for tasks and locations can be performed at different levels of accuracy

during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location ℓ by the program point at which it is created ℓ_{pp}, and each task by the method name executing. They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Both the analysis and the semantics can be made *object-sensitive* by keeping the k ancestor abstract locations (where k is a parameter of the analysis). For the sake of simplicity of the presentation, we assume $k = 0$ in the formalization (our implementation uses $k = 1$).

Example 4. In our working example there are three abstract locations, ℓ_2, ℓ_3 and ℓ_4, corresponding to locations barber, client and chair, created at lines 2, 3 and 4; and six abstract tasks, *sleeps*, *cuts*, *wakeup*, *sits*, *taken* and *isClean*. The following cycle is inferred by the deadlock analysis: $\ell_2 \xrightarrow{11:sleeps} taken \xrightarrow{17:taken} sits \xrightarrow{25:sits} \ell_3 \xrightarrow{24:wakeup} cuts \xrightarrow{12:cuts} \ell_2$. The first arrow captures that the location created at L2 is blocked waiting for the termination of task taken because of the synchronization at L11 of task sleeps. Observe that cycles contain dependencies also between tasks, like the second arrow, where we capture that taken is waiting for sits. Also, a dependency between a task (e.g., sits) and a location (e.g., ℓ_3) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can observed, it is complex to figure out from them why these dependencies arise, and in particular the interleavings scheduled to lead to this situation.

5.2 Guiding Testing towards Deadlock Cycles

Given an abstract deadlock cycle, we now present a novel technique to guide the systematic execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied. Uppercase letters in constraints denote variables to allow representing incomplete information.

Definition 3 (Deadlock-cycle constraints). *Given a state $St = (S, table)$, a deadlock-cycle constraint takes one of the following three forms:*

1. $\exists t_{L,T,PP} \mapsto \langle N, \rho \rangle$*, which means that there exists or will exist an entry of this form in table (time constraint)*
2. $\exists fut(F, L, Tk, p)$*, which means that there exists or will exist a future variable of this form in S (fut constraint)*
3. pending(Tk)*, which means that task Tk has not finished (pending constraint)*

The following function ϕ computes the set of deadlock-cycle constraints associated to a given abstract deadlock cycle.

Definition 4 (Generation of deadlock-cycle constraints). *Given an abstract deadlock cycle* $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$*, and two fresh variables* L_i, Tk_i*,* ϕ *is defined as* $\phi(e_i \xrightarrow{p_i:tk_i} e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_i) =$

$$\begin{cases} \{\exists t_{L_i,Tk_i,\_} \mapsto \langle \_, \mathsf{sync}(p_i,F_i)\rangle, \exists fut(F_i,L_j,Tk_j,p_j)\} \cup \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_j, Tk_j) & \text{if } e_j{=}tk_j \\ \{\mathsf{pending}(Tk_i)\} \ \cup \ \phi(e_j \xrightarrow{p_j:tk_j} \ldots, L_i, Tk_j) & \text{if } e_j = \ell \end{cases}$$

Notation $\mathsf{sync}(p_i, F_i)$ is a shortcut for $p_i{:}F_i.\mathsf{block}$ or $p_i{:}F_i.\mathsf{await}$. Uppercase letters appearing for the first time in the constraints are fresh variables. The first case handles location-task and task-task arrows (since e_j is a task abstraction), whereas the second case handles task-location arrows (e_j is an abstract location). Let us observe the following: (1) The abstract location and task identifiers of the abstract cycle are not used to produce the constraints. This is because constraints refer to concrete identifiers. Even if the cycle contains the same identifier on two different nodes or arrows, the corresponding variables in the constraints cannot be bound (i.e., we cannot use the same variables) since they could refer to different concrete identifiers. (2) The program points of the cycle (p_i and p_j) are used in time and fut constraints. (3) Location and task identifier variables of fut constraints and subsequent time or pending constraints are bound (i.e., the same variables are used). This is done using the 2nd and 3rd parameters of function ϕ. (4) In the second case, Tk_j is a fresh variable since the location executing Tk_i can be blocked due to a (possibly) different task. Intuitively, deadlock-cycle constraints characterize all possible deadlock chains representing the given cycle.

Example 5. The following deadlock-cycle constraints are computed for the cycle in Ex. 4: $\{\ \exists t_{L_1,Tk_1,\_} \mapsto \langle \_, 11{:}F_1.block\rangle,\ \exists\ fut(F_1, L_2, Tk_2, 15),\ \exists t_{L_2,Tk_2,\_} \mapsto \langle \_, 17{:}F_2.await\rangle, \exists\ fut(F_2, L_3, Tk_3, 25),\ \mathsf{pending}(Tk_3),\ \exists t_{L_3,Tk_4,\_} \mapsto \langle \_, 24{:}F_3.block\rangle, \exists fut(F_3, L_4, Tk_5, 12), \mathsf{pending}(Tk_5)\}$. They are shown in the order in which they are computed by ϕ. The first four constraints require *table* to contain a concrete time in which *some* barber sleeps waiting at L11 for a *certain* chair to be taken at L15 and, during another concrete time, this one waits at L17 for a *certain* client to sit at L25. The client is not allowed to sit by the 5th constraint. Furthermore, the last three constraints require a concrete time in which *this* client waits at L24 to get a haircut by *some* barber at L12 and that haircut is never performed. Note that, in order to preserve completeness, we are not binding the first and the second barber. If the example is generalized with several clients and barbers, there could be a deadlock in which a barber waits for a client which waits for another barber and client, so that the last one waits to get a haircut by the first one. This deadlock would not be found if the two barbers are bound in the constraints (i.e., if we use the same variable name). In other words, we have to account for deadlocks which traverse the abstract cycle more than once.

The idea now is to monitor the execution using the inferred deadlock-cycle constraints for the given cycle, with the aim of stopping derivations at states that do not satisfy the constraints. The following boolean function $\mathsf{check}_{\mathcal{C}}$ checks the satisfiability of the constraints at a given state.

Definition 5. *Given a set of deadlock-cycle constraints* $\mathfrak{C}$*, and a state* $St = (S, table)$*,* check *holds, written* $\mathsf{check}_{\mathfrak{C}}(St)$*, if* $\forall t_{L_i,Tk_i,PP} \mapsto \langle N, \mathsf{sync}(p_i, F_i)\rangle \in \mathfrak{C}$*,* $fut(F_i, L_j, Tk_j, p_j) \in \mathfrak{C}$*, one of the following conditions holds:*

1. $\mathsf{reachable}(t_{L_i,Tk_i,p_i}, S)$
2. $\exists t_{\ell_i,tk_i,pp} \mapsto \langle n, \mathsf{sync}(p_i, f_i)\rangle \in table \wedge fut(f_i, \ell_j, tk_j, p_j) \in S \ \wedge$ $(\mathsf{pending}(Tk_j) \in \mathfrak{C} \Rightarrow \mathsf{getTskSeq}(tk_j, S) \neq \epsilon)$

Function reachable checks whether a given task might arise in subsequent states. We over-approximate it syntactically by computing the transitive call relations from all tasks in the queues of all locations in S. Precision could be improved using more advanced analyses. Function getTskSeq gets from the state the sequence of instructions to be executed by a task (which is ϵ if the task has terminated). Intuitively, check does not hold if there is at least a time constraint so that: (i) its time identifier is not reachable, and, (ii) in the case that the interleavings table contains entries matching it, for each one, there is an associated future variable in the state and a pending constraint for its associated task which is violated, i.e., the associated task has finished. The first condition (i) implies that there cannot be more representatives of the given abstract cycle in subsequent states, therefore if there are potential deadlock cycles, the associated time identifiers must be in the interleavings table. The second condition (ii) implies that, for each potential cycle in the state, there is no deadlock chain since at least one of the blocking tasks has finished. This means there cannot be derivations from this state leading to the given cycle, hence the derivation can be stopped.

Definition 6 (Deadlock-cycle guided-testing (DCGT)). *Consider an abstract deadlock cycle* c*, and an initial state* St_0*. Let* $\mathfrak{C} = \phi(c, L_{init}, Tk_{init})$ *with* L_{init}, Tk_{init} *fresh variables. We define DCGT, written* $exec_c(St_0)$*, as the set* $\{d : d \in exec(St_0), \mathsf{deadlock}(St_n)\}$*, where* St_n *is the last state in* d*.*

Example 6. Let us consider the DCGT of our working example with the deadlock-cycle of Ex. 4, and hence with the constraints $\mathfrak{C}$ of Ex. 5. The interleavings table at St_5 contains the entries $t_{ini,main,1} \mapsto \langle 0, return\rangle$, $t_{cl,wakeup,21} \mapsto \langle 1, 24{:}f_0.block\rangle$ and $t_{ba,cuts,12} \mapsto \langle 2, return\rangle\}$. $\mathsf{check}_{\mathfrak{C}}$ does not hold since $t_{L_1,Tk_1,24}$ is not reachable from St_5 and constraint $\mathsf{pending}(Tk_5)$ is violated (task *cuts* has already finished at this point). The derivation is hence pruned. Similarly, the rightmost derivation is stopped at St_{11}. Also, derivations at St_4, St_8 and St_{10} are stopped by function deadlock of Th. 1. Since there are no more deadlock cycles, the search for deadlock detection finishes with this DCGT. Our methodology therefore explores 19 states instead of the 181 explored by the full systematic execution.

Theorem 2 (Soundness). *Given a program* P*, a set of abstract cycles* C *in* P *and an initial state* St_0*,* $\forall d \in exec(St_0)$ *if* d *is a derivation whose last state is deadlock, then* $\exists c \in C$ *s.t* $d \in exec_c(St_0)$*. (The proof can be found in App. A)*

6 Experimental Evaluation

We have implemented our approach within the SYCO tool, a testing tool for *concurrent objects* which is available at http://costa.ls.fi.upm.es/syco, where most

of the benchmarks below can also be found. Concurrent objects communicate via *asynchronous* method calls and use await and block, resp., as instructions for non-blocking and blocking synchronization. This section summarizes our experimental results which aim at demonstrating the effectiveness and impact of the proposed techniques. The benchmarks we have used include: (i) classical concurrency patterns containing deadlocks, namely, *SB* is an extension of the sleeping barber, *UL* is a loop that creates asynchronous tasks and locations, *PA* is the pairing problem, *FA* is a distributed factorial, *WM* is the water molecule making problem, *HB* the hungry birds problem; and, (ii) deadlock free versions of some of the above, named *fX* for the *X* problem, for which deadlock analyzers give false positives. We also include here a peer-to-peer system *P2P*.

Table 6 shows, for each benchmark, the results of our deadlock guided testing (DGT) methodology for finding a representative trace for each deadlock compared to those of the standard systematic testing. Partial-order reduction techniques are not applied since they are orthogonal. This way we focus on the reductions obtained due to our technique per-se. For the systematic testing setting we measure: the number of solutions or complete derivations (column Ans), the total time taken (column T) and the number of states generated (column S). For the DGT setting, besides the time and number of states (columns T and S), we measure the "number of deadlock executions"/"number of unfeasible cycles"/"number of abstract cycles inferred by the deadlock analysis" (column $D/U/C$), and, since the DCGTs for each cycle are independent and can be performed in parallel, we show the maximum time and maximum number of states measured among the different DCGTs (columns T_{max} and S_{max}). For instance, in the DGT for *HB* the analysis has found five abstract cycles, we only found a deadlock execution for two of them (therefore 3 of them were unfeasible), $44s$ being the total time of the process, and $15s$ the time of the longest DCGT (including the time of the deadlock analysis) and hence the total time assuming an ideal parallel setting with 5 processors. Columns in the group **Speedup** show the gains of DGT over systematic testing both assuming a sequential setting, hence considering values T and S of DGT (column T_{gain} for time and S_{gain} for number of states), and an ideal parallel setting, therefore considering T_{max} and S_{max} (columns T^{max}_{gain} and S^{max}_{gain}). The gains are computed as X/Y, X being the measure of systematic testing and Y that of DGT. Times are in milliseconds and are obtained on an Intel(R) Core(TM) i7 CPU at 2.3GHz with 8GB of RAM, running Mac OS X 10.8.5. A timeout of 150s is used. When the timeout is reached, we write $>X$ to indicate that for the corresponding measure we have got X units in the timeout. In the case of the speedups, $>X$ indicates that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X. Also, we write X^* when DGT times out.

Our experiments support our claim that testing complements deadlock analysis. In the case of programs with deadlock, we have been able to provide concrete traces for feasible deadlock cycles and to discard unfeasible cycles. For deadlock-free programs, we have been able to discard all potential cycles and therefore prove deadlock freedom. More importantly, the experiments demonstrate that

	Systematic			DGT (deadlock-per-cycle)					Speedup			
Bm.	**Ans**	T	S	**D/U/C**	T	T_{max}	S	S_{max}	T_{gain}	S_{gain}	T_{gain}^{max}	S_{gain}^{max}
HB	35k	32k	114k	2/3/5	44k	15k	103k	34k	0.73	0.9	2.15	3.33
FA	11k	11k	41k	2/1/3	2k	759	3k	2k	5.5	13.7	15.1	22.2
UL	>90k	>150k	>489k	1/0/1	133	133	5	5	>1.1k	>2.5k	>2.5k	>98k
SB	>103k	>150k	>584k	1/0/1	59	59	23	23	>2.5k	>25k	>2.5k	>25k
PA	>121k	>150k	>329k	2/0/2	42	4	12	6	>3.6k	>27k	>38k	>55k
WM	>82k	>150k	>380k	1/0/2	>150k	>150k	>258k	>258k	1*	1.47*	1*	1.47*
fFA	5k	7k	25k	0/1/1	5k	5k	11k	11k	1.61	2.35	1.61	2.35
fP2P	25k	66k	118k	0/1/1	34k	34k	52k	52k	1.96	2.28	1.96	2.28
fPA	7k	7k	30k	0/2/2	4k	2k	9k	4k	1.75	3.33	3.73	6.98
fUL	>102k	>150k	>527k	0/1/1	410	410	236	236	>1k	>2k	>1k	>2k

Table 1. Experimental results: Deadlock-guided testing vs. systematic testing

our DGT methodology achieves a notable reduction of the search space over systematic testing in most cases. Except for benchmarks HB and WM which are explained below, the gains of DGT both in time and number of states are enormous (more than three orders of magnitude in many cases). It can be observed that the gains are much larger in the examples in which the deadlock analysis does not give false positives (namely, in SB, UL and PA). In general, the generated constraints for unfeasible cycles are often not able to guide the exploration effectively (e.g. in HB and WM). Even in these cases, DGT outperforms systematic testing in terms of scalability and flexibility. Let us also observe that the gains are less notable in deadlock-free examples. That is because, each DCGT cannot stop until all potential deadlock paths have been considered. As expected, when we consider a parallel setting, the gains are much larger.

All in all, we argue that our experiments show that our methodology complements deadlock analysis, finding deadlock traces for the potential deadlock cycles and discarding unfeasible ones, with a significant reduction.

7 Conclusions and Related Work

There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [6, 7] and thread-based programs [11, 13], is based on static analysis techniques. Static analysis can ensure the absence of errors, however it works on approximations (especially for pointer aliasing) which might lead to a "don't know" answer. Our work complements static analysis techniques and can be used to look for deadlock paths when static analysis is not able to prove deadlock freedom. Using our method, we try to find a deadlock by exploring the paths given by our deadlock detection algorithm that relies on the static information.

Deadlock detection has been also studied in the context of dynamic testing and model checking [4, 9, 10, 15], where sometimes has been combined with static information [2, 8]. As regards combined approaches, the approach in [8] first performs a transformation of the program into a trace program that only keeps the instructions that are relevant for deadlock and then dynamic testing is performed on such program. The approach is fundamentally different from ours:

in their case, since model checking is performed on the trace program (that over-approximates the deadlock behaviour), the method can detect deadlocks that do not exist in the program, while in our case this is not possible since the testing is performed on the original program and the analysis information is only used to drive the execution. In [2], the information inferred from a type system is used to accelerate the detection of potential cycles. This work shares with our work that information inferred statically is used to improve the performance of the testing tool, however there are important differences: first, their method developed for Java threads captures deadlocks due to the use of locks and cannot handle wait-notify, while our technique is not developed for specific patterns but works on a general characterization of deadlock of asynchronous programs; their underlying static analysis is a type inference algorithm which infers deadlock types and the checking algorithm needs to understand these types to take advantage of them, while we base our method on an analysis which infers descriptions of chains of tasks and a formal semantics is enriched to interpret them.

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *P*roc. of POPL'14, pages 373–384. ACM, 2014.
2. R. Agarwal, L. Wang and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *H*VC, LNCS 3875. Springer, 2006.
3. E. Albert, P. Arenas and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *F*ORTE'14, Springer.
4. M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In ICST'13, pages 154–163. IEEE, 2013.
5. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *P*roc. POPL'05, pp. 110-121. ACM, 2005.
6. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *F*ORTE'13, LNCS 7892. 2013.
7. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
8. P. Joshi, M. Naik, K. Sen, and Gay D. An effective dynamic analysis for detecting generalized deadlocks. In *P*roc. of FSE'10, pages 327–336. ACM, 2010.
9. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *P*roc. of PLDI'09. ACM, 2009.
10. A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report, 2013.
11. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *P*arallel and Distributed Debugging. ACM, 1991.
12. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *P*roc. of ICSE, pages 386–396. IEEE, 2009.
13. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM* TCS, 1997.
14. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *P*roc. FASE'06, LNCS 3922, pp. 339-356. Springer, 2006.
15. K. Havelund, Using Runtime Analysis to Guide Model Checking of Java Programs, Proceedings of the 7th International SPIN Workshop, Springer-Verlag, 2000.
16. E. Albert, M. Gómez-Zamalloa, et.al. Combining Static Analysis and Testing for Deadlock Detection. http://costa.ls.fi.upm.es/papers/costa/AlbertGI15.pdf.