Analysis of Executable Software Models

Richard Bubel, Antonio Flores Montoya, and Reiner Hähnle
TU Darmstadt, Dept. of Computer Science
bubel|aeflores|haehnle@cs.tu-darmstadt.de

Abstract. In this tutorial we focus on the Abstract Behavioral Modeling (ABS) language, a highly modular, executable modeling language for concurrent systems. We show how three analyses for ABS models are working: resource consumption, deadlock detection, and functional verification. The acceptance of incomplete ABS models together with the capability to analyse them makes ABS extremely useful as a precise modeling language to be used in the design phases of software development.

1 Introduction
Modern software is complex, often runs in a concurrent or distributed environment, and undergoes frequent evolutionary changes driven by rapid changes stemming from business and technological factors. Software is an essential and integral part of most contemporary consumer products, machinery, communication systems, transport systems, etc. The growing ubiquity of software in commodities, but also in safety- and security-critical applications implies that software defects more and more have direct consequences for end users and are of central importance for the acceptance, quality, and safety of many products.

Recall the well-known cost increase for fixing defects during successive software development phases [14]. IBM Systems Sciences Institute estimates that a defect that costs one unit to fix in design, costs 15 units to fix in testing (system/acceptance) and 100 units or more to fix in production (see Fig. 1), and this cost estimation does not even consider the impact cost due to, for example, delayed time to market, lost revenue, lost customers, and bad public relations. Together with the ubiquity of software, the penalty for late discovery of defects makes a very powerful case for software development methods and tools that permit to analyze the consequences of design choices, and possibly erroneous decisions, at an as early stage as possible.

Conventional, informal and semi-formal notations, such as the UML or feature diagrams, however, are not rich and formal enough to admit simulation, automated analysis, or rapid prototyping. It is with this gap in mind that in the past years there has been a lot of interest in executable modeling languages.

In this tutorial we focus on the Abstract Behavioral Specification (ABS) language [25,1], a highly modular, executable modeling language for concurrent

* Partly funded by the EU project FP7-610582 Envisage.
Fig. 1. Relative costs to fix software defects for static infrastructure (source: IBM Systems Sciences Institute). The columns indicate the phase of the software development at which the defect is found and fixed.

systems that exhibit a high degree of product variability. ABS is a rich, object-oriented language with strong typing, strong encapsulation, abstract data types, a simple, but powerful concurrency model. What sets it apart from mainstream programming languages are three aspects: first, ABS comes with a formal, operational semantics [17]; second, ABS has been carefully designed so as to make automated analyses of various kinds feasible; third, ABS models can be partially specified. Because of the first two features, it is possible to construct a range of automatic and semi-automatic analysis tools for the full ABS language. In the present tutorial we show how three analyses of particular importance are working: resource consumption (Sect. 3.1), deadlock detection (Sect. 3.2), and functional verification (Sect. 3.3). The acceptance of incomplete ABS models together with the capability to analyse them makes ABS extremely useful as a precise modeling language to be used in the design phases of software development.

To make this chapter self-contained, we include a very concise introduction into the ABS language in Sect. 2, however, we strongly recommend to read the tutorial [23] as a background. To make the content of this chapter manageable, we focus on two analysis methods for ABS, but we stress that a whole range of tools is available for ABS [34]. In the present volume, the interested reader can find more information on an alternative approach to deadlock analysis in the chapter by Laneve et al., on test generation in the chapter of Albert et al., and on runtime assertion checking in the chapter of de Boer et al.

2 Setting the Context: Abstract Behavioral Modeling

2.1 The Abstract Behavioral Specification (ABS) Language

In this section we briefly introduce the Abstract Behavioral Specification (ABS) language [25, 1]. The text is based on the ABS introduction given in [35]. For readers unfamiliar with ABS, we recommend the tutorial [23].
ABS is an abstract, executable, object-oriented modeling language [25]. It has been designed as a modeling language that is in particular well equipped for the modeling needs of distributed systems with a high degree of variability.

Formal treatment of ABS models is possible, because the ABS modeling language is properly defined in terms of a formal SOS-style semantics. In particular, all design decisions are carefully crafted to ensure that ABS models are amenable to formal analyses. ABS is under active development and current research targets modeling and analysis of cloud-based services with respect to service contracts [16].

Fig. 2 shows the layered architecture of ABS. The base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an object-oriented imperative layer similar but much simpler than Java. The concurrency model of ABS is two-tiered: at the lower level it is similar to that of JCoBox [32] that generalizes the concurrency model of Creol [26] from single concurrent objects to concurrent object groups (COGs). COGs encapsulate synchronous, multi-threaded, shared state computation on a single processor. On top of this is an actor-based model with asynchronous calls, message passing, active waiting, and future types. An essential difference to thread-based concurrency is that task scheduling is cooperative, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows to write concurrent programs in a much less error-prone way than in a thread-based model and makes ABS models suitable for static analysis. Specifically, the ABS concurrency model excludes race conditions on shared data.

Local contracts and assertions allow to specify a wide variety of functional properties about ABS programs in a Design-by-Contract [28] style. The top layer Delta Modeling Language (DML) adds delta-oriented programming [31] to ABS. Although being a central feature in ABS, delta modeling and variability-aware
analyses are out of scope for this paper. The contribution of Clarke et al. in this volume contains more information on variability modeling.

2.2 ABS Example

In this tutorial we use a simple banking system as a running example. Fig. 3 shows some of its interfaces.

```
interface Account {
  Int getAid();
  Int deposit(Int x);
  Int withdraw(Int x);
  Bool transfer(Int amount, Account target);
}

interface DB {
  Unit insertAccount(Account a);
  Maybe<Account>getAccount(Int aid);
}
```

Fig. 3. Banking example: Interfaces

The interface `Account` models a bank account with the expected services such as deposit and withdrawal of money. The interface `DB` models the bank database used to manage accounts. In particular, it provides means to query for an account using its unique account number.

In Fig. 4 an implementing class of interface `Account` is shown. A major design decision is that the balance of accounts must never be negative. Hence, in case of a withdrawal it is checked, whether the account has a sufficient balance to perform the withdrawal. Otherwise, no money is withdrawn and the method returns the unchanged balance.

Fig. 5 shows how class `DatabaseImpl` implements the `DB` interface. The method `getAccount(Int)` implements the lookup for a given account number `aid` as follows: it iterates through the list of all accounts managed by the database. For each managed account it looks up the account number via an asynchronous method call `a!getAid()`. In case of success, the found account is returned, otherwise `null` is returned.

3 Analysis Methods

3.1 Resource Analysis

Automatic resource analysis attempts to infer safe upper bounds on the amount of resources that might be consumed by a program or model during its execution
class AccountImpl(Int aid, Int balance) implements Account {
    Int getAid() { return aid; }

    Int deposit(Int x) {
        balance = balance + x;
        return balance;
    }

    Int withdraw(Int x) {
        if (balance - x >= 0) {
            balance = balance - x;
        }
        return balance;
    }

    Bool transfer(Int amount, Account target) { ... }
}

Fig. 4. Banking example: Account implementation

as a function of its input variables. A resource can be any magnitude that we are interested to measure for a given model execution. Time or memory consumption are typical examples of resources. There is an extensive literature on program resource analysis, both for the functional and imperative paradigm [5, 22, 21, 24, 10, 36, 33, 15]. However, most approaches are focused on sequential programs and do not treat concurrent programs. This is not a coincidence, given that concurrency adds both inherent and accidental complexity to a resource analysis. Inherent complexity stems mainly from the increased non-determinism that comes with concurrency. Accidental complexity is a consequence of the choice of concurrency models that derive from low-level primitives that are prevalent in current mainstream programming languages. Collaborative scheduling, as realized in ABS, reduces the inherent complexity of the analysis as it reduces the number of possible interleavings that might occur. On the other hand, the use of future variables and synchronization on guards reduces its accidental complexity. As a consequence, in contrast to languages such as Java or C/C++, it is possible to automatically analyze concurrent ABS models and obtain resource consumption upper bounds for many interesting and realistic examples.

Basic approach. We introduce the basic approach to resource analysis of sequential programs [6, 5] that we later adapt to concurrent ABS models. Before analyzing a program we abstract away from all information that is not relevant for resource consumption. An abstract representation that turns out to be useful is based on cost equations. Cost equations are a specific kind of non-deterministic recurrence relations enriched with a constraint \( \varphi \) that relates the variables that appear in the cost equation and imposes applicability conditions on it. A cost
equation \( c(\bar{x}) = e, \phi \) represents a fragment of code (typically a method or a loop) with integer variables \( \bar{x} \), where \( e \) represents the cost of executing the fragment of code as a function of \( \bar{x} \) and might contain references to other cost equations.

```java
class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int n = length(as);
        Int cnt = 0;
        while (cnt < n) {
            Account a = nth(as,cnt);
            Fut<Int>idFut = a.getAid();
            Int id=idFut.get;
            if (aid == id) {
                result = a;
            }
            cnt = cnt+1;
        }
        return result;
    }
    ...
}
```

**Fig. 5.** Simplified example of bank database query

**Example 1.** The (simplified) cost equations of the method `getAccount` from Fig. 5 are:

- \( \text{getAccount}(as,aid) = 3 + \text{length}(as) + \text{while}(0,n,aid,as) \) \( n = as \)
- \( \text{while}(cnt,n,aid,as) = 4 + \text{nth}(as,cnt) + \text{getAid}(a) + \text{if}(cnt,n,aid,a) + \text{while}(cnt + 1,n,aid,as) \) \( cnt < n \)
- \( \text{while}(cnt,n,aid,as) = 0 \) \( cnt \geq n \)
- \( \text{if}(cnt,n,aid,a) = 1 \) \( a = aid \)
- \( \text{if}(cnt,n,aid,a) = 0 \) \( a \neq aid \)

The cost equations of length, nth and getAid have been omitted.

The cost expression \( e \) is obtained by applying a cost model to the ABS model. Intuitively, a cost model maps each instruction to a cost. The choice of the cost model determines the resources that we want to observe. For example, if our cost model maps every instruction to a cost of 1, we will infer an upper bound on the number of executed instructions. Or we could assign a different cost to each `new C` instruction according to the type of object created (and 0 to any other instruction) to measure the heap memory consumption.
Example 2. The cost model applied in our example counts the number of assignments. The cost equation

\[
\langle \text{getAccount}(as, aid) = 3 + \text{length}(as) + \text{while}(0, n, aid, as), \quad n = as \rangle \tag{1}
\]

contains the constant 3, because of the assignments \texttt{Account result = null;}\,; \texttt{Int n = length(as);}\,; \texttt{and Int cnt = 0;} \texttt{in getAccount.}

To obtain the constraint \(\varphi\) of a cost equation, each variable is abstracted to its “size” according to a chosen size measure and the instructions are substituted by constraints that represent the effect of the instructions on the size of the variables. The set of constraints obtained in this way for a code fragment are then conjoined to a single predicate \(\varphi\). A typical size measure for arrays and lists is their length. The constraint \(\varphi\) can be enriched with invariants generated using abstract interpretation techniques.

Example 3. The constraint of cost equation (1) reflects the use of size measures and invariants. The list \(as\) has been abstracted to its length and through invariant generation techniques we obtain that the result value \(n\) of \(\text{length}(as)\) is the length of the list \(as\), that is \(n = as\).

There are multiple techniques to solve systems of cost equations [4, 9, 11]. In general, the strongly connected components (SCCs) in a system of cost equations are determined and incrementally solved. For each SCC, we look for a ranking function that bounds the number of its possible iterations. Then, we approximate the cost of each iteration as a function of the initial variables.

Example 4. We compute the cost of while following the approach of [4]. Assume the cost of \(\text{nth}(as, cnt)\) is \(cnt\) and the cost of \(\text{getAid}(a)\) and \(\text{if}(cnt, n, aid, a)\) are 0 and 1, respectively. The cost of one iteration of while is \(4 + cnt + 1\). The value of \(cnt\) changes in each iteration, but we can use the invariant \(cnt \leq n\) to infer that \(cnt\) is bounded by \(n\). Now we can approximate any iteration by \(5 + n\). Finally, the function \(n - cnt\) is a valid ranking function of while, because it is always non-negative and it decreases with each iteration. A valid upper bound of while is, therefore, \((n - cnt) \times (5 + n)\).

Concurrency. ABS’s concurrency model poses additional challenges to resource analysis [3]. During the execution of a task, other interleaving tasks can modify the values of the shared variables (that is, object fields). This has to be taken into account when generating a suitable abstraction of ABS models. A safe approximation consists in “forgetting” all the information related to object fields every time when an interleaving might occur (at \texttt{await} and \texttt{suspend} instructions). This loss of information can reduce the precision of the analysis or even prevent obtaining upper bounds.

Example 5. In Fig. 6 we consider a small modification of the code in Fig. 5. We have removed the auxiliary variable \(n\) and we do not block the complete database each time we want to obtain an account’s id. In the cost equation abstraction of
class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int cnt = 0;
        while (cnt < length(as)) {
            Account a = nth(as, cnt);
            Fut<Int>idFut = a!getAid();
            await idFut?;
            Int id = idFut.get;
            if (aid == id) {
                return result;
            }
            cnt = cnt + 1;
        }
    }
    ...
}

Fig. 6. Bank database query with concurrency

instruction await idFut?; we lose the information about the object’s fields. The resulting cost expressions of while are:

while(cnt, n, aid, as) = 4 + length(as) + nth(as, cnt) + getAid(a) +
+ if(cnt, n, aid, a) + while(cnt + 1, n, aid, as')
while(cnt, n, aid, as) = length(as)

In these new cost equations we are not able to find a ranking function, because as can vary at every iteration. Therefore, no upper bound is found.

This approximation can be improved using class invariants. A class invariant in ABS is a predicate on the object fields that holds not only at the beginning and end of each method, but also at every release point.

Example 6. If we can infer the class invariant \( as \leq as_{max} \), we can include this invariant after each release point:

\[
\begin{align*}
\text{while}(cnt, n, aid, as) &= 4 + length(as) + nth(as, cnt) + \text{getAid}(a) + \\
&\quad \text{if}(cnt, n, aid, a) + \text{while}(cnt + 1, n, aid, as') \\
\text{cnt} &< as \\
\text{cnt} &\geq as
\end{align*}
\]

With that invariant, we can find the ranking function \( as_{max} - cnt \) and obtain an upper bound.
A more advanced technique for proving termination and for inferring upper bounds of loops with interleavings was presented in [8]. That technique follows a rely-guarantee style of reasoning. Assume we have a loop whose termination proof fails because of the information lost at the release points. First, we assume that the shared variables are not modified at the release points, but we do not assume any initial value. Given this assumption we try to prove termination again using standard techniques. If we fail to prove termination, the interleavings were not the cause of the failure. If we succeed, we know that without interleavings the loop terminates. We can also conclude that if the number of interleavings that modify the fields involved in the termination proof is finite, then the loop will also terminate. As we did not assume any initial value on the fields to prove termination, after any modification, the loop is still terminating. If the modifications are finite, the overall system will terminate.

In addition, one has to prove the assumption, that is, the number of times the fields are modified during execution of the loop is finite. To this end, examine the program points that modify fields. These points can be filtered through a May-Happen-in-Parallel (MHP) analysis [7] (see also Sect. 3.2) to keep only those points that can possibly be executed during the execution of the loop. Then try to prove that the remaining program points are executed a finite number of times by proving termination of all the loops that can reach these program points. If we find a circular dependency, that is, the need to prove termination of a loop to prove its own termination, the process terminates with a failure.

Cost Centers. Distributed systems are usually composed of multiple machines, each with its own resources. But traditionally the output of a resource analysis consists only of a single cost expression of the overall cost. This is not appropriate for distributed systems. It is more interesting to obtain separate cost expressions for each distributed component. This can be achieved with the notion of cost centers [3].

A cost center is a part of a distributed system with resources whose consumption we want to measure independently from other parts of the system. For example, in ABS cost centers might correspond to COGs or single objects. We can generate cost equations where each part of the cost is multiplied by a constant that represents the cost center where that cost is incurred. For example, the cost equation \( C(x) = 2 \cdot c_1 + 3 \cdot c_2 \), \( \varphi \) represents code that consumes 2 resource units in cost center \( c_1 \) and 3 units in cost center \( c_3 \). Once a set of cost equations parameterized with cost centers \( c_1, c_2, \ldots, c_n \) is obtained, we can compute the resources consumed by a cost center \( c_j \). We set \( c_i = 1 \) and \( c_j = 0 \) for every \( j \neq i \) and solve the cost equations as usual.

### 3.2 Deadlock Analysis

As explained in Sect. 2.1, ABS models use a high-level concurrency model that does not deal explicitly with primitives such as locks or semaphores. This allows us to implement static deadlock analyses that are both precise and efficient. In general, deadlock situations are produced when a concurrent model reaches a
state in which one or more tasks are waiting for each others’ termination and none of them can make any progress. The combination of blocking (get) and non-blocking (await) operations in ABS can result in complex deadlock situations.

To realize a deadlock analysis we have to identify the elements that can contribute to a deadlock situation and their mutual dependencies. In the case of ABS, these elements can be tasks and COGs. There can be three kinds of dependencies among tasks and COGs:

1. *task-task* dependencies, when a task waits for the termination of another task with a non-blocking operation (await);
2. *COG-task* dependencies, when a task waits for the termination of another task but keeps the COG’s lock (using get);
3. *task-COG* dependencies that occur between each task and the COG they belong to.

The set of these dependencies form a *dependency graph*, where the nodes of the graph are the tasks and COGs involved. A deadlock can occur if, at some point during the execution, there is circular dependency in the active dependencies at that point. Given a concrete state, we can extract a dependency graph. If such graph is cyclic, the state is a deadlock state.

**Example 7.** Given the following code:

```java
class AImp() implements A {
  Unit syncMessage(A x, String m) {
    Fut<Unit> f = x.recv(m);
    f.get
  }

  Unit AsyncMessage(A x, String m) {
    Fut<Unit> f = x.recv(m);
    await f;
  }

  Unit recv(String m){
  }
}

{ 
  A a1 = new cog AImp();
  A a2 = new cog AImp();
  a1.syncMessage(a2,'ping');
  a2.AsyncMessage(a2,'ping');
}
```

The corresponding dependency graph is:
One possible approach for statically detecting deadlock situations is to infer a safe, abstract dependency graph. That is, we want to infer a dependency graph such that any cycle in the dependency graph of any concrete execution can be mapped to a cycle in the abstract dependency graph. If the abstract graph has no cycles, no cycle will be possible in any concrete execution of the model.

We can approximate the dependency graphs with a points-to analysis, similar to the one of [29]. A points-to analysis generates a set of abstract objects that belong to abstract COGs forming an abstract configuration. Each object is abstracted by a sequence of allocation points of a fixed length that determines the precision of the analysis. For each abstract object \( o \) and method in that kind of object \( m \), we have an abstract task \( o.m \). The points-to analysis also provides information on which objects may be pointed to by each reference at any program point. Here, future variables are considered as special references that point to abstract tasks. The dependency graph can be constructed as follows: The nodes of the graph are the abstract COGs and the abstract tasks formed from the method names and abstract objects. The edges can be obtained by examining the points-to information of the future variables at the synchronization points (the \texttt{await} and \texttt{get} instructions).

An important source of imprecision is the fact that we infer a single dependency graph that “covers” all the possible concrete graphs. In the abstract graph there might be dependencies that form a cycle but that cannot be active simultaneously in any concrete execution state. Such a situation would generate a false positive. We can discard some of these unfeasible cycles with a May-Happen-in-Parallel (MHP) analysis [7]. A MHP analysis tells us, given two program points, whether there can be any concrete state in which those two points are being executed in parallel. A dependency cycle is feasible if all the synchronization points that generated its dependencies can happen in parallel.

This approach has shown to be efficient and precise enough for many practical cases. The major source of imprecision is the abstraction performed by the points-to analysis which fixes the set of possible abstract objects beforehand. In particular, all objects created inside a loop are abstracted to a single abstract object. Whenever there are dependencies among these objects’ tasks, we will get spurious deadlock alerts. The latter are handled better by contract-based approaches, such as the one of Cosimo et al [19, 20] (see also the Chapter by Laneve at al. in this volume).
3.3 Deductive Verification

For real-world programming languages like Java, deductive verification of distributed and concurrent programs is hard. A major reason for this are concurrency models that are not well-defined, platform-dependent or too liberal. These weaknesses cause a proliferation of the possible interleavings that have to be checked for a given property. Hence, much research effort has been directed towards techniques that allow to restrict the number of possible interleavings, for example, symmetry reductions.

As explained in Sect. 2.1, the ABS language was designed around a concurrency model whose analysis stays manageable. Shared memory communication is only possible within a concurrent object group (COG), for which ABS permits only cooperative scheduling. Hence, all interleaving points occur syntactically explicit in an ABS program in form of an await statement which releases control. Communication between different COGs (which are executed in parallel on distributed nodes) is restricted to message passing.

The limitations of the ABS concurrency model makes it possible to define a compositional specification and verification method. This is essential for being able to scale verification to non-trivial programs, because it is possible to specify and verify each ABS method separately, without the need for a global invariant. During formal verification of ABS, we do not model threads or process queues explicitly, and hence, stay in an essentially sequential setting. This makes it possible to largely reuse a well-understood specification approach for sequential, imperative programs. We follow the Design-by-Contract [28] paradigm with an emphasis on specification of interface and class invariants.

The ABS verification method instantiates a combination of the rely-guarantee and assumption/commit paradigms [27, 30]. The workflow is as follows: For each interface and each implementing class appropriate invariants are specified:

**Interface invariants** express mostly restrictions on the control-flow, i.e., constraints on the order of asynchronous method calls.

**Class invariants** are mainly used to relate the state of an object to the local history of the system. The history is a sequence of events such as method invocations, method completions, or object creations. For instance, a method invocation event is implicitly generated and recorded in the object-local history whenever a method is called asynchronously.

To verify an ABS model we prove that for each class an arbitrarily chosen object preserves its interface and its class invariants. The compositionality of our method then gives the guarantee that these invariants are preserved by all objects of the system.

To specify history properties we use a formalisation of histories that was developed in [18]. For the purpose of this tutorial, we restrict ourselves to the four event types depicted in Fig. 7.

1. **Object s invokes asynchronously method m on object r**. This asynchronous invocation results in the creation of a future and is also recorded as an *invocation event* in the history of the caller object s.
(2) Once the method invocation is scheduled for execution in \( r \), an *invocation reaction event* is created and recorded in the history of the callee \( r \).

(3) After the execution of method \( m \) completes and resolves the future, an accompanying *completion event* is created and recorded in the history of \( r \).

(4) When the future gets finally queried for the return value (usually by the invoking object) a *completion reaction event* is added to the history of the caller \( s \).

\[ \text{Fig. 7. History events and when they occur} \]

Specification and verification of ABS models is done in ABS dynamic logic (ABS DL). ABS DL is a typed first-order logic with the addition of a box modality: Let \( \phi \) denote an ABS DL formula, and \( p \) be a sequence of executable ABS statements, then

\[ [p] \phi \] (spoken: box \( p \phi \)) is an ABS DL formula with the (informal) meaning: If \( p \) terminates then \( \phi \) hold in its final state.

In addition, ABS DL uses updates (taken from [13]) to capture state changes. An *elementary update* has the form \( x := t \) where \( x \) is a program variable and \( t \) a term. Updates can be applied to formulas or terms: Let \( u \) be an update and \( \xi \) a term (formula), then \( \{u\} \xi \) is a term (formula).

**Example 8.** Given a program variable \( i \) and the formula \( i > 0 \). Then evaluating the formula

\[ \{i := 3\}(i > 0) \]

in a program state \( s \) means that \( i > 0 \) is evaluated in a state \( s' \) which coincides with \( s \) on all program variables except for \( i \), which has the value 3. The meaning of an update is identical to the meaning of an assignment whose only side-effect is the actual update of the value stored in the location on the left-hand side. The above formula is this equivalent to

\[ [i=3];(i > 0) \].
To express properties of a system in terms of histories, ABS DL uses a dedicated, globally defined program variable `history`, which contains the union of all object-local histories as a sequence of events. The history events themselves are elements of datatype `Event`, which defines for each event type a constructor function. For instance, an invocation event is represented as `invocEv(s, r, fut, m, args)` where `s` is the caller, `r` the callee, `fut` the created future, `m` the asynchronously called method and `args` the method arguments.

In addition to the history formalization as a sequence of events, there are a number of auxiliary and convenience predicates that allow to express common properties concerning histories. For example, predicates like `wfHist(History)`, `beginsWith(History, Event)`, `endsWith(History, Event)`, etc., are used to specify wellformedness of histories, etc.

To verify that an ABS program satisfies a specified property, a Gentzen-style sequent calculus is used. A sequent is a data structure of the form:

\[ \phi_1, \ldots, \phi_m \Rightarrow \psi_1, \ldots, \psi_n \]

which has the same meaning as the formula

\[ \bigwedge_{i \in \{1 \ldots m\}} \phi_i \rightarrow \bigvee_{j \in \{1 \ldots n\}} \psi_j. \]

A sequent rule

\[ \text{name} \]

\[ \text{premise} \]

\[ s_1 \ldots \ldots \ldots \ldots s_n \]

\[ s \]

\[ \text{conclusion} \]

\((s, s_i, i \in \{1 \ldots n\} \text{ are sequents})\) has a name, a premise consisting of a possibly empty sequence of sequents and a conclusion. A sequent rule is called correct if the validity of the premise implies the validity of the rule’s conclusion. An axiom is a sequent rule without premise.

A sequent proof is a tree where each node is labelled with a sequent and there exists a sequent rule \(r\) for each inner node such that the conclusion of \(r\) matches the node’s sequent and the rule’s premises match the sequents of the node’s children. A branch (of the proof tree) is called closed if the last rule application was an axiom. A proof is called closed if and only if all its branches are closed.

The sequent calculus as realized in ABS DL essentially simulates a symbolic interpreter for ABS. The assignment rule for a local program variable is:

\[
\begin{align*}
\text{assign} \quad & \quad \Gamma \Rightarrow \{v := e\} [\text{rest}] \phi, \Delta \\
\frac{}{\Gamma \Rightarrow [v=e; \text{rest}] \phi, \Delta}
\end{align*}
\]

where \(v\) is a local program variable and \(e\) a pure (side effect free) expression. The rule rewrites the formula by moving the assignment from the program into an update. During symbolic execution the updates accumulate in front of the modality containing the executed program. Once the program to be verified has been completely executed and the modality is empty, these updates are applied to
the formula after the modality, resulting in a pure first-order formula (assuming there are no nested modalities). An example for a rule that causes the proof tree to split is

\[
\text{ifSplit} \quad \Gamma, e \doteq \text{True} \Rightarrow [p; \text{rest}]\phi, \Delta \quad \Gamma, e \doteq \text{False} \Rightarrow [q; \text{rest}]\phi, \Delta
\]

where for each branch of the conditional statement a corresponding proof branch is created. Each of the two branches has to be considered and closed to prove that the property \(\phi\) holds after the ABS program terminates.

We conclude this section with the rules for asynchronous method invocation and the \texttt{await} statement:

\texttt{asyncMC}
\[
\Gamma \Rightarrow o \neq \text{null} \land \text{wfHist}(\text{history}), \Delta
\]
\[
\Gamma \Rightarrow \{U\}(\text{futureUnused}(frc, \text{history}) \rightarrow \\
   \{fr := frc || \text{history} := \text{append}((\text{history}, \text{invocEv}(\text{this}, o, frc, m, e)))\})[\text{rest}]\phi)
\]
\[
\{U\}[r = o!m(\text{args}); \text{rest}]\phi
\]

In case of an asynchronous method invocation the proof splits into two branches: the first branch (displayed on top) ensures that the callee is not \texttt{null} and that the history is wellformed. The second branch introduces a new constant \(frc\) which represents the future (placeholder for the method’s return value). The left side of the implication ensures that the future is new and it has not yet been used (\texttt{futureUnused}) and updates the history by appending the invocation event for the asynchronous method call. Afterwards, execution continues with the remaining program \texttt{rest}. The sequent rule for the \texttt{await} statement is:

\texttt{awaitComp}
\[
\Gamma \Rightarrow C\text{inv}(C)(\text{heap}, \text{history}, \text{this}), \Delta
\]
\[
\Gamma \Rightarrow \{\text{heap} := \text{newHeap} || \\
   \text{history} := \text{append}((\text{history}, \text{append}(\text{newHist, compREv}(...))))\})
\]
\[
(C\text{inv}(C)(\text{heap}, \text{history}, \text{this}) \land \text{wfHist}(\text{history}) \rightarrow [\text{rest}]\phi), \Delta
\]
\[
\Gamma \Rightarrow \text{await } r?; \text{rest}]\phi, \Delta
\]

where \texttt{newHist, newHeap} are fresh Skolem constants; \(C\) is the class in which the ABS code in the premise’s modality is executed.

The \texttt{await} statement releases control allowing other threads to take over. Once the \texttt{await} guard is satisfied (here: the future is resolved), the waiting thread can be rescheduled. As control of the COG is released by the currently executed code, we must ensure that a state has been reached in which the invariant of class \(C\) is satisfied, because the continuing thread will rely on it. The fulfillment of that class invariant is checked by the first branch.

The second branch assumes that the \texttt{await} condition is satisfied and continues the execution in a state where the completion reaction event has been appended to the extended history. This means that the value of the \texttt{history} variable before execution of the \texttt{await} statement has been some event sequence
(modeled with the Skolem constant newHist), representing those events that occurred between control release and control resume. In our rely-guarantee-based setting, we can safely assume that upon resume of control, the class invariant has been established by the previous thread and holds again. But the heap might have been changed and all previously accumulated knowledge about it must be removed. This is achieved by assigning to the heap an unknown value (modeled with the Skolem constant newHeap).

4 Application Examples

4.1 Resource Analysis

We explore the possibilities of the different cost models, size measure, and cost center definitions. We will analyze the example from Fig. 8. The resource and termination analysis is part of the SACO tool [2] available at http://costa.is.fi.upm.es/web/saco.php. Once the SACO plugin has been installed, please create an ABS project with the code of our example. To analyze the program, we select the method getAccount in the Outline view. Then, we select SACO->Analyze with SACO. A dialog will appear showing the different analyses available in SACO. We check Resource Analysis and click on Analyze. Unfortunately, the result we obtain contains the term c(maximize failed) which indicates a failure in the maximization process. This is, because even if there are no concurrent interleavings, we need an invariant for the initial value of the field as. So we add an invariant at the beginning of the method:

```java
[as <= max(as)]
Account getAccount(Int aid) {
  Account result = null;
  ...
}
```

Once the invariant is added we obtain a valid upper bound. Instead of analyzing directly, we can select SACO->Analyze with SACO, check Resource Analysis and click on Configure+Analyze. Now we can select the parameters of the analysis. Some of the options are:

Cost model: indicates the type of resource that we are interested in measuring. Some of the cost models are: Steps (counts the number of executed instructions), Tasks (counts the number of asynchronous calls to methods), Memory (measures the size of the created data structures).

Cost centers: allows to decide whether we want to use cost centers or not. If we decide to use cost centers, we can choose between class and object. The option class associates a cost center to each class, whereas object associates a cost center to each abstract object inferred in the points-to analysis.

Size abstraction: allows choosing how data structures are abstracted into an integer number. Two possibilities are provided: Size, which counts all nodes in the structure, and Depth, which counts the length of the longest path.
Now we analyze the number of tasks that are created during the execution of `getAccount` in total. We set the cost model to `Tasks`, no cost centers and size abstraction `Depth` (Our main data structure is a list and `Depth` corresponds to the length of the list). We obtain that the number of tasks is $\max(as)$ which corresponds to the number of calls to `getAid`.

Next we perform an analysis with cost centers. We select the option cost center `class` and the cost model `Steps`. The result is:

$$12 + 6 \cdot \text{nat}(\max(as) - 1) + \text{nat}(\max(as) - 1) \cdot (20 + 9 \cdot \text{nat}(\max(as) - 2))$$

within cost-center 'DBImpl'

$$\text{nat}(\max(as) - 1) \text{within cost-center 'AccountImpl'}$$

Here we can see how the cost in `AccountImpl` is linear but is quadratic in `DBImpl`. This quadratic cost is due to the function `nth` that has linear cost and is executed a linear number of times. Knowing that, we could try to improve the method to avoid the quadratic cost.

**Rely-Guarantee Termination.** We try to prove termination of the example with interleavings in Fig. 6. To apply the rely-guarantee method, we need a complete model with a main block (termination depends on which other methods can be executed in parallel). We add the the following main block:

```plaintext
{ 
  Account a;
  DB db = new cog DBImpl();
  Int max = 10;
  Int i = 1;
  while(i <= max){
    a = new cog AccountImpl(i,0);
    Fut<Unit> aFut = db!insertAccount(a);
    await aFut?;
    i = i+1;
  }
  db!getAccount(3);
}
```

In this main block we create a database, then add 10 new accounts with account ids ranging from 1 to 10, and finally we query the database with the account 3.

To analyze the resulting program, we select the main block in the Outline view, select SACO->Analyze with SACO, check Termination Analysis and click on Analyze. The result is a list of strongly connected components (SCCs) and the information whether they are terminating or not. In this case, all the SCCs turn out to be terminating. The termination of `getAccount` depends on `as`. However, when `getAccount` is executed, all the `insertAccount` calls must have terminated. That is detected by the MHP analysis and thus termination is proven.

If we remove the instruction `await aFut?`, this is not the case any more. Several instances of `insertAccount` might execute in parallel with `getAccount`.
But we can prove termination of the loop in the main block, and this implies that as can be modified in parallel only a finite number of times (10 times) and, therefore, `getAccount` is still guaranteed to terminate.

### 4.2 Deadlock analysis

```java
//Module and Interface declarations have been omitted

class ClientI(Server server) implements Client {
    Config config = null;
    Unit setConfig(Config co) {
        config=co;
    }
    Unit syncSend(String m) {
        // await config!=null;
        Fut<Unit> f = server!recv(m);
        f.get;
    }
}

class ServerI implements Server{
    Config co = null;
    Unit ini(Client client) {
        co = new ConfigI();
        Fut<Unit> f = client!setConfig(co);
        f.get;
    }
    Unit recv(String message) {
    }
}

{  
    Server s = new cog ServerI();
    Client c = new cog ClientI(s);
    s!ini(c);
    c!syncSend("hello");
}
```

**Fig. 8.** Client-server deadlock example

We illustrate the behavior of the analysis with the example code of Fig. 8. The example has a `main` method (line 44) that creates a server and a client. Then, it initializes the server with a reference to the client at line 47. The method `ini()` (line 35) creates a `Config` object and passes it to the client using the method
setConfig() (line 23). The server should not do anything until the client has received the configuration so it waits holding the lock at line 38. Finally, the main method calls syncSend() in line 48. Method syncSend() (line 26) sends a message to the server by calling recv() (line 40) and blocks the client until recv() is completed in line 30.

A deadlock can occur if syncSend() (line 26) starts before setConfig() (line 23). The server will stay blocked at line 38 waiting for setConfig() to finish. At the same time the client will stay blocked at line 30 waiting for recv(). Neither setConfig() nor recv() is able to start as their COGs are blocked by other methods.

The deadlock analysis proceed as follows. First, is uses the points-to information to identify the objects, COGs and tasks that can be created: c (the client), s (the server) and their respective tasks: c.setConfig, c.syncSend, s.ini, and s.recv. Second, it identifies the synchronization points and extracts their dependencies: line 30 generates c \(\rightarrow\) s.recv and c.syncSend \(\rightarrow\) s.recv; line 38 generates s \(\rightarrow\) c.setConfig and s.ini \(\rightarrow\) c.setConfig; also the dependencies from each task to its COG: s.recv \(\rightarrow\) s, s.ini \(\rightarrow\) s, c.setConfig \(\rightarrow\) c, and c.syncSend \(\rightarrow\) c.

![Deadlock dependency graph](image-url)

Fig. 9. Deadlock dependency graph of example from Fig.8

In the thus constructed dependency graph (see Fig. 9), we look for cycles. There is one cycle: c \(\rightarrow\) s.recv \(\rightarrow\) s.ini \(\rightarrow\) c.setConfig \(\rightarrow\) c. Finally, we check whether all program points involved in the cycle can happen in parallel using the MHP analysis. In this case, all the involved points (line 30, line 40, line 38 and line 23) can happen in parallel to each other and the tool will report the deadlock cycle. If we uncomment line 27, setConfig() is forced to finish before proceeding to line 30. Therefore, no deadlock is possible. The dependency graph is the same, but the MHP analysis reports that line 30 and line 23 now cannot happen in parallel and the cycle is discarded.

This deadlock analysis is part of the SACO tool (See Sec. 4.1). Lets use the Eclipse plugin interface to analyze the example from Fig. 8. Once the SACO
plugin has been installed, we create a ABS project with the code of our example. In order to analyze the program, we select the Main Block in the Outline view. Then, we select SACO->Analyze with SACO. A dialog will appear showing the different analyses of SACO. Check the option Deadlock Analysis and click Analyze. Shortly after, a report of the possible deadlocks will appear in the Eclipse console (See Fig. 10) and the synchronization instructions involved in the deadlocks will appear highlighted. Again, if we uncomment line 27 and repeat the analysis, we will see a new message in the console indicating that the program has no deadlocks.

```
cog ServerI(45,main) blocked in object ServerI(45,main) at ServerI.ini Line 38
   (Waiting for)
   \/  task ClientI.setConfig in object ClientI(46,main) in cog ClientI(46,main)
   (MHP)
   \/  cog ClientI(46,main) blocked in object ClientI(46,main) at ClientI.syncSend Line 30
   (Waiting for)
   \/  task ServerI.recv in object ServerI(45,main) in cog ServerI(45,main)
```

**Fig. 10.** Output of the deadlock analysis for the example of Fig. 8

### 4.3 Deductive Verification

We illustrate verification of ABS models along some examples. The account types supported by the banking system example are not allowed to be in debt, i.e., their balance must always be non-negative. To verify that our ABS model implements this policy, we need to specify the property as an invariant of class AccountImpl in Fig. 4. Invariants for interfaces and classes are specified in a separate file whose suffix is .inv as follows:

```java
\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {
  nonNegativeBalance : Account.AccountImpl {
    int::select(heapSV, self, Account.AccountImpl::balance) >= 0
  };
}
```

The keyword `invariants` opens a section wherein invariants can be specified. Its parameters declare program variables that can be used to refer to the history (historySV), the heap (heapSV), and the current object (self, similar as Java’s this). These program variables can be used in the specification of class invariants.

The section declares an invariant with the name nonNegativeBalance for class AccountImpl. The class invariant states that the value of field balance for
the current object must be non-negative. The built-in function `int::select` is
the standard heap selection function for return type `Int`.

Loading the problem in KeY-ABS opens the proof obligation selection dialog
shown in Fig. 11. On selection of the proof obligation *Preserves Class Invariant*
for method `withdraw(Int)` of class `AccountImpl`, a proof obligation of the
following (slightly simplified) form is generated:

\[
\{ \text{history} := \text{append}(\text{history}, \text{invocRev}(\ldots)) \}
\quad ((\text{CInv}(\text{heap}, \text{history}, \text{self}) \land \text{wfHist}(\text{history})) \rightarrow [\text{mb}; \text{CInv}(\ldots)])
\]

where `mb` denotes the body of method `withdraw(Int)`. In this example the proof
obligation can be proven automatically with a few steps.

Fig. 11. Proof-Obligation selection dialog

The attempt to prove that the invariant is preserved as well by method
`deposit(Int)` fails with one open goal. Inspecting the goal reveals that the
method cannot be proven for negative arguments of `deposit(Int)`. This is not
an issue for method `withdraw` which has an explicit check, but for `deposit(Int)`
negative arguments need to be excluded using either a precondition or an invariant.
An invariant is more appropriate, because it reflects the design decision that
accounts never run a negative balance. Moreover, an invariant lets one reuse the
restriction also in other contexts.

```c
\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {
    amountOfDepositNonNegative : Account.AccountImpl {
        \forall Event ev; ( ...
```
\textbf{forall} int \( i; ( \ i \geq 0 \ \& \ i < \) seqLen(historySV) \rightarrow \\
(\ ev = \ Event::seqGet(historySV, i) \ \& \\
( isInvocationEv(ev) \ | \ isInvocationREv(ev) ) \ \& \\
getMethod(ev) = \ Account.Account::deposit#ABS.StdLib.Int \rightarrow \\
int::seqGet(getArguments(ev), 0) \geq 0 ) )

This invariant ensures that method \texttt{deposit(Int)} is in any event history always invoked with a non-negative argument by inspecting the associated invocation (reaction) events. With this additional invariant we can close the proof for \texttt{deposit(Int)}, requiring to instantiate the second quantifier in the invariant once by hand.

As a final example, we specify how the value of field \texttt{balance} of class \texttt{Account} relates to the history: it always coincides with the value returned by the most recent call of the \texttt{deposit(Int)} or \texttt{withdraw(Int)} method. We specify this property as follows:

\begin{verbatim}
\textbf{invariants} (Seq historySV, Heap heapSV, ABSAnyInterface self) {

balanceConsistent : Account.AccountImpl {
\textbf{forall} Event ev;
( ev = Event::seqGet(historySV, seqLen(historySV) - 1) \& \\
ev = compEv(self, getFuture(ev), getMethod(ev), getResult(ev)) \& \\
( getMethod(ev) = Account.Account::withdraw#ABS.StdLib.Int | \\
getMethod(ev) = Account.Account::deposit#ABS.StdLib.Int ) ) \\
\rightarrow \\
getResult(ev) = int::select(heapSV,self, \\
Account.AccountImpl::balance) 

};
}
\end{verbatim}

This invariant can be proven automatically for the methods \texttt{deposit(Int)} and \texttt{withdraw(Int)}. The proofs of all invariants combined require 954 proof steps with only two user interactions for method \texttt{deposit(Int)} and 1700 proof steps for method \texttt{withdraw(Int)} with no user interactions (see Fig. 12).

\section{Conclusion and Future Perspectives}

We discussed three complementary analyses techniques for the ABS modeling language: deadlock detection, resource consumption, and deductive verification. None of the analyses in this tutorial would have been possible with the same degree of automation and precision in implementation languages such as Java or C/C++. It is a crucial insight that ABS was developed from the start with analyzability in mind. As the ABS examples demonstrate (and, even more so, industrial case studies [34]), it is nevertheless possible to create rich and realistic software models.
The three presented analyses differ in difficulty of usage and in precision: easiest to use is the deadlock detection analysis, which is fully automatic and does not require any configuration. If the analysis finds a problem, the call chain leading to the potential deadlock is shown and the involved statements are highlighted in the Eclipse IDE. The deadlock analysis is correct, i.e., when no deadlocks are reported, the analyzed ABS program is deadlock-free. But, as a consequence of abstraction and over-approximation, not all reported deadlocks need actually occur, so one has to carefully check the analysis report to reject false positives.

The resource consumption analysis requires that a cost model was specified or at least an a priori specified cost model needs to be selected. The actual analysis is again fully automatic and the derived costs for the ABS model are shown. The analysis might, however, not always return with a result. If it returns with an upper bound, then this is sound, that is, no concrete run of the ABS model will exceed the computed worst case. It is, however, possible that no concrete run reaches the upper bound, that is, the analysis might not be tight.

Deductive verification clearly is the most difficult to use analysis presented in this paper. It requires to specify invariants of the system and the verification process requires some amount of user interaction. Both activities require considerable expertise with formal specification and verification. On the positive side, the deductive verification is precise and highly expressive with respect to the properties that can be specified. It allows to verify data dependent, functional properties of ABS models. An in-depth discussion of the trade-offs of various verification scenarios can be found in [12].
References


