Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems*

Crystal Chang Din[1], Olaf Owe[1] and Richard Bubel[2]
[1]*University of Oslo, Oslo, Norway*
[2]*Technische Universität Darmstadt, Darmstadt, Germany*

Keywords: Runtime Assertion Checking, Formal Verification, Concurrency, Distributed Systems, Tools.

Abstract: We investigate the usage of a history-based specification approach for concurrent and distributed systems. In particular, we compare two approaches on checking that those systems behave according to their specification. Concretely, we apply runtime assertion checking and static deductive verification on two small case studies to detect specification violations, respectively to ensure that the system follows its specifications. We evaluate and compare both approaches with respect to their scope and ease of application. We give recommendations on which approach is suitable for which purpose as well as the implied costs and benefits of each approach.

1 INTRODUCTION

Distributed systems play an essential role in society today. However, quality assurance of such systems is non-trivial since they depend on unpredictable factors. It is highly challenging to test distributed systems after deployment under different relevant conditions. These challenges motivate frameworks combining precise modeling and analysis with suitable tool support.

Object orientation is the leading framework for distributed systems, recommended by the RM-ODP (International Telecommunication Union, 1995). The paradigm of concurrent objects communicating by *asynchronous method calls* appears as a promising framework to combine object-orientation and distribution in a natural manner. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* (Baker Jr. and Hewitt, 1977, Halstead Jr., 1985, Liskov and Shrira, 1988) improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, one enables other objects to get method results directly from future objects. We consider a core language following these principles, based on the ABS language (HATS, 2011).

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components (Hoare, 1985). At any point in time the communication history abstractly captures the system state. In fact, traces are used in semantics for full abstraction results (e.g., (Jeffrey and Rathke, 2005, Ábrahám et al., 2009)). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties (Alpern and Schneider, 1985).

In this work we present and compare a runtime assertion checker with a verification system/theorem prover for concurrent and distributed systems using object-orientation, asynchronous method calls and futures. Communication histories are generated through the execution and are assumed wellformed. The modeling language is extended such that users can define software behavioral specification (Hatcliff et al., 2012), i.e., invariants, preconditions, assertions and postconditions, inline with the code. We provide the ability to specify history-based properties, which are verified during simulation. Although by runtime assertion checking, we gain confidence in the quality of programs, correctness is still not fully guaranteed for

*Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services. (http://www.envisage-project.eu)

Chang Din C., Owe O. and Bubel R..
Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems.
DOI: 10.5220/0004877804800487
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 480-487
ISBN: 978-989-758-007-9

all runs. Formal verification may instead show that a program is correct by proving that the code satisfies a given specification. We choose KeY (Beckert et al., 2007) as our formal verification tool since it is a highly automated theorem prover, and with support for ABS. We extended KeY with extra rules for dealing with history-based properties. At the end we compare the differences and challenges of using these two approaches.

Paper overview. Section 2 introduces and explains the core language, Section 3 presents (1)the reader-writer example and (2)the publisher-subscriber example, Section 4 formalizes the observable behavior in the distributed systems, Section 5 shows the result of runtime assertion checking on the examples (1) and (2), Section 6 shows the result of theorem proving on the examples (1) and (2), Section 7 compares runtime assertion checking with theorem proving, Section 8 discusses related work and we then close with remarks about future work.

2 THE CORE LANGUAGE

Our core language is presented in Fig 1. It includes basic statements for first order futures, taken from *ABS* (HATS, 2011), which is an executable modeling language designed with hindsight to the modeling, formal analysis and code generation of concurrent and distributed systems.

Methods are organized in classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, and defines fields $\overline{w}$, an optional initialization block s, and methods $\overline{M}$. There is read-only access to class parameters $\overline{cp}$ as well as method parameters. A method definition has the form $m(\overline{x})\{\texttt{var}\ \overline{y};\ s;\ \texttt{return}\ e\}$, ignoring type information, where $\overline{x}$ is the list of parameters, $\overline{y}$ an optional list of *method-local variables*, s a sequence of statements, and the value of e is returned upon termination. As in the Creol language (Johnsen and Owe, 2007), a reference to the caller is an implicit parameter denoted caller.

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

$$
\begin{array}{lcl}
Cl & ::= & \texttt{class}\ C([T\ cp]^*)\ \{[T\ w\ [:= e]^?]^*\ [s]^?\ M^*\} \\
M & ::= & T\ m([T\ x]^*)\ \{[\texttt{var}\ [T\ x]^*]^?\ s\ ;\ \texttt{return}\ e\} \\
T & ::= & C \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Void} \mid \mathit{Fut}{<}T{>} \\
v & ::= & x \mid w \\
e & ::= & \texttt{null} \mid \mathsf{this} \mid v \mid cp \mid f(\overline{e}) \\
s & ::= & v := e \mid fr := v!m(\overline{e}) \mid v := e.\mathsf{get} \mid \texttt{skip} \\
 & & \mid \texttt{await}\ e \mid \texttt{await}\ e? \mid \texttt{assert}\ e \\
 & & \mid \texttt{while}\ (e)\ \{s\} \mid \texttt{if}\ (e)\ \{s\}\ [\texttt{else}\ \{s\}]^? \\
 & & \mid v := \texttt{new}\ C(\overline{e}) \mid s;s
\end{array}
$$

Figure 1: Core language syntax. Expressions e are side-effect free, $\overline{e}$ is a (possibly empty) expression list. $[\backsim]^*$ and $[\backsim]^?$ denote repeated and optional parts.

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

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

Class instances are concurrent, encapsulating their own state and processor, similarly to the actor model (Hewitt et al., 1973). Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. A release point may cause the active process to be suspended, allowing the processor to resume other (enabled) processes. Note that a process, as well as the initialization code of an object, may make self calls to recur-

sive methods with release points thereby enabling interleaving of active and passive behavior. The core language considered here ignores features orthogonal to futures, including interface encapsulation and local synchronous calls.

As in *ABS*, abstract data types are supported and in this paper we will use the following notation for sets and sequences. The empty set is denoted Empty, addition of an element x to a set s is denoted $s+x$, the removal of an element x from a set s is denoted $s-x$, and the cardinality of a set s is denoted $\#s$. Similarly, the empty sequence is denoted Nil, addition of an element x to a sequence s is denoted $s \cdot x$, the removal of all x from a sequence s is denoted s/x, and the length of a sequence s is denoted $\#s$. Indexing of the ith element in a sequence s is denoted $s[i]$ (assuming i is in the range $0...\#s-1$). Membership in a set or sequence is denoted $\in$.

3 EXAMPLES

We illustrate runtime assertion checking and theorem proving for the programs in the core language via two examples: a fair version of the reader/writer example and a publisher/subscriber example. The first example shows how we verify the class implementation by relating the objects state with the local communication history. The second example shows how we achieve compositional reasoning by proving the order of the local history events for each object.

3.1 The Reader-Writer Example

We assume given a shared database db, which provides two basic operations read and write. In order to synchronize reading and writing activity on the database, we consider the class RWController, see Fig. 2. All client activity on the database is assumed to go through a single RWController object. The RWController provides read and write operations to clients and in addition four methods used to synchronize reading and writing activity: openR, closeR, openW and closeW. A reading session happens between invocations of openR and closeR and writing between invocations of openW and closeW. Several clients may read the database at the same time, but writing requires exclusive access. A client with write access may also perform read operations during a writing session. Clients starting a session are responsible for closing the session. Clients have the interface *CallerI* (omitted here).

Internally in the class, the attribute readers contains a set of clients currently with read access and

```
class RWController implements RWinterface{
  DB db := new DataBase();
  Set<CallerI> readers := Empty;
  CallerI writer := null; Int pr := 0;

  Void openR(){ await writer = null;
     readers := readers + caller;}

  Void closeR(){
     readers := readers − caller;}

  Void openW(){ await writer = null;
     writer := caller;
     readers := readers + caller;}

  Void closeW(){ await writer = caller;
     writer := null;
     readers := readers − caller;}

  String read(){ await caller ∈ readers;
     pr := pr + 1;
     Fut<String> fr := db!read(key);
     await fr?; String s := fr.get;
     pr := pr - 1; return s;}

  Void write(Int key, String value){
     await caller=writer && pr=0 &&
          readers − caller = Empty;
     Fut<Void> fr:=db!write(key,value);
     fr.get;} }
```

Figure 2: Implementation of class RWController.

writer contains the client with write access. If there is no writer, a client gains write access by execution of openW. A client may thereby become the writer even if readers is non-empty. Nevertheless, the controller ensures that reading and writing *activity* cannot happen simultaneously on the database. The client with write access will eventually be allowed to perform write operations since all active readers (other than itself) are assumed to end their sessions at some point. For readability reasons, we declare local variables in the middle of a statement list, and omit the (redundant) return statement of *Void* methods.

3.2 The Publisher-subscriber Example

In this example clients may subscribe to a service, while the service object is responsible for generating news and distributing each news update to the subscribing clients. To avoid bottlenecks when publishing events, the service delegates publishing to a chain of *proxy* objects, where each proxy object handles a bounded number of clients. The implementation of the classes Service and Proxy is given in Fig. 3. Again, interfaces are omitted.

The example applies the future concept by letting

```
class Service(Int limit,NewsProducerI np)
                    implements ServiceI{
  ProducerI prod; ProxyI proxy;
  ProxyI lastProxy;

  {prod := new Producer(np);
   proxy := new Proxy(limit,this);
   lastProxy := proxy; this!produce();}

  Void subscribe(ClientI cl){
    Fut<ProxyI>last := lastProxy!add(cl);
    lastProxy := last.get;}
  Void produce(){
    Fut<News> fut := prod!detectNews();
    proxy!publish(fut);}}

class Proxy(Int limit, ServiceI s)
                    implements ProxyI{
  List<ClientI> myClients = Nil;
  ProxyI nextProxy;

  ProxyI add(ClientI cl){ ... }
  Void publish(Fut<News> fut){
    News ns := fut.get; ...
    client!signal(ns); ...
    nextProxy!publish(fut);}}
```

Figure 3: Implementation of class Service and Proxy.

the service object delegate publishing of news updates to the proxies without blocking while waiting for the result of the news update. This is done by the sequence $fut := prod!detectNews(); proxy!publish(fut)$. Furthermore, the calls on *add* are blocking, however, this is harmless since the implementation of *add* may not deadlock and terminates efficiently.

4 OBSERVABLE BEHAVIOUR

The observable behavior of a system is described by *communication histories* over observable events (Hoare, 1985). Since message passing is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, namely the generating object. Assume an object o calls a method m on object o' with input values $\overline{e}$ and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $\langle o \rightarrow o', u, m, \overline{e} \rangle$ generated by o. An *invocation reaction event* $\langle o \twoheadrightarrow o', u, m, \overline{e} \rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow o', u, m, e \rangle$. This event reflects that u is resolved with return value e. The *fetching event* $\langle o \twoheadleftarrow, u, e \rangle$ is generated by o when fetching the value of the resolved future. Since future identities may be passed to other objects, e.g, o'', that object may also fetch the future value, reflected by the event $\langle o'' \twoheadleftarrow, u, e \rangle$, generated by o''. The *object creation event* $\langle o \uparrow o', C, \overline{e} \rangle$ represents object creation, and is generated by o when creating a fresh object o'.

For a method call with future u, the ordering of events is described by the regular expression $\langle o \rightarrow o', u, m, \overline{e} \rangle \cdot \langle o \twoheadrightarrow o', u, m, \overline{e} \rangle \cdot \langle \leftarrow o', u, m, e \rangle [\cdot \langle \_ \twoheadleftarrow, u, e \rangle]^*$ for some fixed o, o', m, $\overline{e}$, e, and where _ denotes arbitrary values. Thus the result value may be read several times, each time with the same value, namely that given in the preceding future event. A communication history is *wellformed* if the order of communication events follows the pattern defined above, the identities of the generated future is fresh, and the communicating objects are non-null.

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

A *global invariant* can be obtained as a conjunction of class invariants for all objects in the system, adding wellformedness of the global history (Dovland et al., 2005).

5 RUNTIME ASSERTION CHECKING

We implement the runtime assertion checking using the Maude backend of ABS. The *ABS* compiler frontend, which takes a complete *ABS* model of the software system as input, checks the model for syntactic and semantic errors and translates it into an internal representation. There are various compiler backends. Maude is a tool for executing models defined in rewriting logic. The Maude back-end takes the internal representation of *ABS* models and translates them to rewriting systems in the language of Maude for simulation and analysis.

The history-explicit semantics in Maude is implemented by adding a global history reflecting all events that have occurred in the execution. The local histories are obtained by projection. We extend the *ABS* language with annotations for specifying pre/-post conditions and invariants. And underlying history functions are implemented.

5.1 Specifying and Verifying the Reader-writer Example

We define a class invariant I for RWController:

$$I \triangleq Writers(\mathcal{H}) = \{\text{writer}\} - \text{null}$$

where $\mathcal{H}$ denotes the local history. This illustrates how the values of class attributes may be expressed in terms of observable communication. The invariant I expresses that if the set of writers retrieved from the history by function $Writers(h)$ is empty then the class attribute *writer* is null, otherwise it contains only one element which is the same as the non-null *writer*. The definition of $Writers : Seq[Ev] \to Set[Obj]$ is:

$$\begin{aligned}
&Writers(\text{Nil}) \triangleq \text{Empty}\\
&Writers(h \cdot \langle \leftarrow \text{this}, fr', \text{openW}, _\rangle) \triangleq\\
&\qquad Writers(h) + irev(h, fr').caller\\
&Writers(h \cdot \langle \leftarrow \text{this}, fr', \text{closeW}, _\rangle) \triangleq\\
&\qquad Writers(h) - irev(h, fr').caller\\
&Writers(h \cdot \textbf{\textit{others}}) \triangleq Writers(h)
\end{aligned}$$

where ***others*** matches all events not matching any of the above cases. The function $irev(h, fr')$ extracts the invocation reaction event, containing the future fr', from the history h. The caller is added to the set of writers upon termination of openW, and the caller is removed from the set upon termination of closeW.

Implementation. The global history is not transparent in *ABS* programs, therefore we provide for each history function a built-in predicate in the *ABS* language without explicit history argument. For instance, the built-in predicate $getWriters()$ returns the result of $Writers(h)$ from the interpreter.

The concrete formulation of I as given to the runtime assertion checker is

```
I: compareSet(getWriters(),
       Empty + writer - null)
```

where `compareSet(`s_1`,`s_2`)` returns true if the set s_1 is equal to the set s_2.

5.2 Specifying and Verifying the Publisher-subscriber Example

In the publisher-subscriber example, we consider object systems based on the classes Server and Proxy of Fig. 3. We may state properties, like:

For every *signal* invocation from a proxy *py* to a client *c* with news *ns*, the client must have *subscribed* to a service *v*, which must have issued a *publish* invocation with a future *u* generated by a *detectNews* invocation, and then the proxy *py* must have received news *ns* from the future *u*.

This expresses that when clients get news it is only from services they have subscribed to, and the news is resulting from actions of the server. Since this property depends on pattern matching, we define an algebraic data type `Event` in *ABS*. We show below the definition of two of the five constructors of the `Event` type:

```
data Event =
InvocEv(Any callee, Any fut,
          String method, Any arg)|
InvocrEv(Any callee, Any fut,
          String method, Any arg)|...;
```

This `Event` type is used to define class invariants, to be verified at runtime for each related object. The generating object is redundant in the local invariants and therefore is omitted from the events.

Since there is no super type in the current *ABS* language, we cannot define the type of each argument. Consequently, to specify the value of the arguments in the *ABS* events is currently not straight forward. We overcome this limitation by defining an algebraic data type `Any`:

```
data Any = O | F | AR | any ;
```

Letting all arguments in the events be of type `Any` except method names and class names which are `String`. The constructors of type `Any` are `any`, a special constant used as a place-holder for any expression, and `O`, `F`, and `AR`, are artificial constants used as place-holders for object identities, future identities, and arguments, respectively, to simulate pattern matching in history functions. The constants `O`, `F`, and `AR`, are used in patterns where a pattern variable occurs more than once, letting all occurrences of `O` match the same value (and similarly for `F` and `AR`), whereas each occurrence of `any` matches any value. For our example, it is enough to define one constructor for each kind. To identify different variables of the same kind, more constructors would be needed, e.g. O_1 and O_2 for object identities. Now we may define a class invariant I for *Service*:

```
I: has(InvocEv(any,any,"add",any)) =>
 isSubseq(list[InvocEv(any,any,"add",AR),
         InvocrEv(any,any,"subscribe",AR)])
```

The predicate `has` checks the existence of an event in the local history. A list `list[a,b,c]` declares the order of the events where (surprisingly) event `a` is the latest. The predicate `isSubseq` checks if the list of events is a subsequence of the local history. The search for a subsequence by `isSubseq` starts from the latest event and continues backwards until finding the first match. In this way, the proved property is prefix-closed by runtime assertion checking. For

instance, the invariant *I* expresses that if the local history of the Server object has an invocation event which reflects a call to a method `add` on some object, there should exist an invocation reaction event with a method name `subscribe` in the prefixed local history and by pattern matching these two events contain the same argument `AR`. If run-time checking gives that *I* holds for the current state, the Server object always receives a client before sending the client to the Proxy.

6 THEOREM PROVING USING KEY

In this section we describe our experiences with the verification of some properties of the reader-writer and the publisher-subscriber examples. The KeY theorem prover (Beckert et al., 2007) is a deductive verification system for sequential Java programs. For this case study, we used a variant of the system which supports reasoning about *ABS* programs. The underlying logic is a first-order dynamic logic for *ABS* (ABSDL) similar to (Ahrendt and Dylla, 2012). For an *ABS* program `p` and an ABSDL formula ϕ, the formula $[p]\phi$ is true if and only if the following holds: *if* `p` *terminates then in its final state* ϕ *holds*. Given an *ABS* method *m* with body *mb* and a class invariant *I*, the ABSDL formula $I \rightarrow [mb]I$ expresses that the method *m* preserves the class invariant.

6.1 Formalizing and Verifying the Reader-writer Example

Formalization of the invariants and proof-obligations for the purpose of verification proves harder than for runtime assertion checking. Parts of the reasons are purely technical and are due to current technical shortcomings of the KeY tool which can and will be overcome relatively easily, e.g., absence of a general set datatype, automation of reasoning about sequences and similar. Other reasons are more deeply rooted in a basic difference between runtime assertion checking and verification. To a certain extent runtime assertion checking can take advantage of a closed system view. A closed system view allows to safely assume that certain interleavings (await statements) will never happen. This allows to simplify the formalization of some invariants considerably, in contrast to verification where we take an open world assumption, and in addition, have to consider all possible runs.

We take here a closer look at the formalization of the invariant *I* from Section 5.1. Invariant *I* states that at most one writer may exist at any time and that if a writer exists then it is the one set by the most recently completed `openW` invocation. In a first step, we define some auxiliary predicates and functions that help us to access the necessary information: First we define the function `getWriter` which takes the local history as argument and returns a sequence of all writers for which a successful completed `openW` invocation exists that has not yet been matched by a completed `closeW` invocation. The axiomatization in ABSDL (slightly beautified) looks as follows:

$$
\begin{aligned}
&\forall h \forall w \big(\\
&\quad w \neq \texttt{null} \wedge \exists \texttt{i}(\texttt{getWriters}(\texttt{h}).\texttt{get}(\texttt{i}) = \texttt{w}) \\
&\qquad \Leftrightarrow \\
&\quad \exists e(\texttt{isFutEv}(\texttt{e}) \wedge \texttt{e} \in \texttt{h} \wedge \texttt{getMethod}(\texttt{e}) = \texttt{openW} \wedge \\
&\qquad w = \texttt{getCaller}(\texttt{getIREv}(\texttt{h}, \texttt{getFut}(\texttt{e}))) \wedge \\
&\qquad \forall e'(\texttt{later}(\texttt{e}', \texttt{e}, \texttt{h}) \wedge \texttt{isFutEv}(\texttt{e}') \rightarrow \\
&\qquad\qquad \texttt{getMethod}(\texttt{e}') \neq \texttt{closeW})) \quad \big)
\end{aligned}
$$

where *h* is the local history, `isFutEv` tests if the given event is a *future event*, and `getIREv` returns the *invocation reaction event* from a given history and future. The other functions should be self-explanatory. We can now state our version of *I* for an object `self`:

$$
\begin{aligned}
&\texttt{length}(\texttt{getWriters}(\texttt{h})) \leq 1 \wedge \\
&\texttt{self.writer} = (\texttt{length}(\texttt{getWriters}(\texttt{h})) = 0 \ ? \\
&\texttt{null} : \texttt{getWriters}(\texttt{h}).\texttt{get}(0))
\end{aligned}
$$

Note that the formalization here is stronger than the one used in runtime assertion checking as we allow at most one writer in the list of writers, i.e., we disallow also that the same writer calls (and completes) `openW` repeatedly. This stronger invariant is satisfied by our implementation.

An important lemma we can derive from the definition of `getWriters` is that it is independent of events other than *future events* and *invocation reaction events* for `openW` and `closeW`. This allows us to simplify the history at several places and to ease the proving process.

6.2 Formalizing and Verifying the Publisher-subscriber Example

The formalization and verification of the publisher subscriber example is inherently harder than that for the reader writer example. The reason is that the properties to be specified focus mainly on the structure of the history. Further, in presence of control releases the history is extended by an unspecified sequence of events. In contrast to runtime assertion checking, we can mostly only rely on the invariants to regain knowledge about the history after a release point. This entails also that we need to actually specify additional invariants expressing what could not have happened in between, e.g., certain method invocations. We formalized the property similar to the runtime assertion

approach using an axiomatization of loose sequences.

For runtime assertion checking it was possible to use pattern matching to express the invariant of Section 5.2. On the logic level, we have to use quantification to achieve the same effect. This impairs at the moment automation as the efficiency of quantifier instantiations decreases rapidly with the number of nested quantifiers.

7 COMPARISON

In this section we discuss the main differences in the scope and application between runtime assertion checking and formal verification. We highlight in particular the difficulties we faced in the respective approaches.

Runtime-assertion checking shares with testing that it is a method to detect the presence of bugs and gives us confidence in the program's quality. Formal verification can instead prove that a program is correct, i.e., the program code satisfies the given specification.

A closer look at the considered specifications reveals that for runtime assertion checking, we check whether a method satisfies its pre- and postcondition at invocation reaction and future resolving time, respectively. An assertion failure was reported if these were not satisfied. In verification, we face the following additional challenge: The caller of a method can only ensure that the precondition holds at the time of the invocation event. But the caller has no control over the system itself and thus cannot ensure that the property still holds when the invoked method is scheduled at the callee side. Possible solutions to this problem are to ensure that once the precondition is proved, it is satisfied until and including the moment when the method is scheduled; a different approach would be to restrict preconditions to express only history and state independent properties about the method parameters. An analogous problem exists for postconditions.

Similarly, formal verification is harder when compared to assertion checking as in the latter we are only concerned with a closed system, namely, the one currently running. This puts less demands on the completeness of specifications as the number of reachable states is restricted by the program code itself. In formal verification we have to consider all states that are not excluded by the specification. For instance, in runtime assertion checking, it is not necessary to specify that the same object does not call openW twice without a call to closeW in between, while this has to specified explicitly for verification purposes.

The need for strong invariant specifications arises in particular when dealing with await statements which release control. During verification, the history is extended by an unspecified sequence of events before continuing the execution after those release points. Almost any knowledge about the new extended history has then to be provided by the invariants.

Further, it turned out that the specifications relied heavily on quantification and recursively defined functions/properties. This makes automation of the proof search significantly more difficult, and finally, required many direct interactions with the prover. In contrast to the symbolic execution in verification, specifications need to be executable in runtime assertion checking such that using quantifiers is not an option. For our purposes, pattern matching is instead applied in this place (when the same place holder appears more than once). In addition, the tool used for formal verification is still work-in-progress and not yet on par with the degree of automation KeY achieves when verifying Java programs.

8 RELATED WORK

Behavioral reasoning about distributed and object-oriented systems is challenging. Moreover, the gap in reasoning complexity between sequential and distributed, object-oriented systems makes tool-based verification difficult in practice. A survey of these challenges can be found in (Ahrendt and Dylla, 2012).

The present approach follows the line of work based on communication histories to model object communication events in a distributed setting (Hoare, 1985, Dahl, 1977). Objects are concurrent and interact solely by method calls and futures. By creating unique references for method calls, the *label* construct of Creol (Johnsen and Owe, 2007) resembles futures. Verification systems capturing Creol labels can be found in (Dovland et al., 2005, Ahrendt and Dylla, 2012). However, a label reference is local to the caller and cannot be shared.

A compositional reasoning system for asynchronous methods in ABS with futures is introduced in (Din et al., 2012). In this work, we realize the reasoning system (Din et al., 2012) in two approaches: runtime assertion checking and theorem proving in KeY (Beckert et al., 2007). The article (Hatcliff et al., 2012) surveys behavioral interface specification languages with a focus toward automatic program verification. A prototype of the verification system (Ahrendt and Dylla, 2012) based on the two-event semantics (Dovland et al., 2005) has been implemented

in KeY (Beckert et al., 2007) but requires more complex rules than the present one.

9 CONCLUSIONS AND FUTURE WORK

In this paper we specified two small *concurrent and distributed* programs and checked their adherence to the specification using two different approaches: *runtime assertion checking* and *deductive verification*. We were in particular interested in how far the use of histories allows us to achieve a similar support for distributed system as state-of-the-art techniques achieve for sequential programs. The results are positive so far: runtime assertion checking is nearly on par with that of a sequential setting. Deductive verification does harder, but some of the encountered issues stem from the current early state of the used tool, where support for reasoning about histories is not yet automatized to a high degree. In the future we intend to improve on the automation of the used tool.

REFERENCES

Ábrahám, E., Grabe, I., Grüner, A., and Steffen, M. (2009). Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming, 78(7):491–518.

Ahrendt, W. and Dylla, M. (2012). A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309.

Alpern, B. and Schneider, F. B. (1985). Defining liveness. *Information Processing Letters*, 21(4):181–185.

Baker Jr., H. G. and Hewitt, C. (1977). The Incremental Garbage Collection of Processes. In *Proc. of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, USA. ACM.

Beckert, B., Hähnle, R., and Schmitt, P. H., editors (2007). *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer.

Dahl, O.-J. (1977). Can program proving be made practical? In Amirchahy, M. and Néel, D., editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, France.

Din, C. C., Dovland, J., and Owe, O. (2012). Compositional reasoning about shared futures. In et al, G. E., editor, *Proc. Intl. Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 94–108. Springer.

Dovland, J., Johnsen, E. B., and Owe, O. (2005). Verification of concurrent objects with asynchronous method calls. In *Proc. IEEE Intl. Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press.

Halstead Jr., R. H. (1985). Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538.

Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. (2012). Behavioral interface specification languages. *ACM CS*, 44(3):16:1–16:58.

HATS (2011). Full ABS Modeling Framework (Mar 2011). Deliverable 1.2 of project FP7-231620 (HATS), available at http://www.hats-project.eu.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proc. 3rd international conference on Artificial intelligence*, pages 235–245.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall.

International Telecommunication Union (1995). Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC.

Jeffrey, A. S. A. and Rathke, J. (2005). Java Jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, volume 3444 of *LNCS*, pages 423–438. Springer.

Johnsen, E. B. and Owe, O. (2007). An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling, 6(1):35–58.

Liskov, B. H. and Shrira, L. (1988). Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In Wise, D. S., editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press.