UNIVERSITY OF OSLO
Department of Informatics

A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems

Research Report 435

Crystal Chang Din

Olaf Owe

Richard Bubel

ISBN 978-82-7368-400-4
ISSN 0806-3036

November 2013

A Comparison of 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, Norway
[2]Technische Universität Darmstadt, Germany

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 distributed systems is non-trivial since they depend on unpredictable factors. , such as different processing speeds of independent components. 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 concurrent and distributed systems, recommended by the RM-ODP [International Telecommunication Union, 1995]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [Ahern and Yoshida, 2007]. Concurrent objects communicating by *asynchronous method calls* appears as a promising framework to combine object-orientation and distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while

[*]crystald@ifi.uio.no
[†]olaf@ifi.uio.no
[‡]bubel@cs.tu-darmstadt.de

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, Liskov and Shrira, 1988, Halstead Jr., 1985, Yonezawa et al., 1986] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to get method results directly from the future object. We consider a core language following these principles, based on the ABS language [HATS, 2011]. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [Hoare, 1985]. At any point in time the communication history abstractly captures the system state [Dahl, 1992, Dahl, 1987]. 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 both state-based and 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 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)reader writer example and (2)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

For the purposes of this paper, we consider a core object-oriented language with futures, presented in Fig 1. It includes basic statements for first order futures,

Cl	::=	**class** $C([T\ cp]^*)$ $\{[T\ w\ [:= e]^?]^*\ [s]^?\ M^*\}$	class definition
M	::=	$T\ m([T\ x]^*)$ $\{$**var** $[T\ x]^*]^?\ s$; **return** $e\}$	method definition
T	::=	$C \mid Int \mid Bool \mid String \mid Void \mid Fut{<}T{>}$	types
v	::=	$x \mid w$	variables (local or field)
e	::=	**null** $\mid$ this $\mid v \mid cp \mid f(\overline{e})$	pure expressions
s	::=	$v := e \mid fr := v!m(\overline{e}) \mid v := e$.get	statements
		$\mid$ **await** $e \mid$ **await** $e? \mid$ **assert** $e \mid v :=$ **new** $C(\overline{e})$	
		$\mid$ **while** (e) $\{s\}$ $\mid$ **if** (e) $\{s\}$ [**else** $\{s\}]^?$ $\mid$ **skip** $\mid s; s$	

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

inspired by *ABS* [HATS, 2011]. 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 $m(\overline{x})\{$**var** $\overline{y}$; s; **return** $e\}$, ignoring type information, where $\overline{x}$ is the list of parameters (as in the Creol language [Johnsen and Owe, 2007], a reference to the caller will be an implicit parameter and the language guarantees that caller is non-null), $\overline{y}$ an optional list of *method-local variables*, s a sequence of statements, and the value of e is returned upon termination.

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 communicated to third party objects, and these may then fetch the future value. A future value may be fetched several times, possibly by different objects. In this manner, shared futures provide an efficient way to distribute method call results to a number of objects.

A future variable fr is declared by $Fut{<}T{>}\ fr$, 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 **await** $fr?$ releases the process until the future fr is resolved. The query statement $v := fr$.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 **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: $Fut{<}Int{>}\ fr := v!m();$ **await** $fr?;$ $Int\ x := fr.\mathsf{get}$, corresponding to a non-blocking method call, whereas $Fut{<}Int{>}\ fr := v!m();\ Int\ x := fr.\mathsf{get}$ 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 recursive methods with release points thereby enabling interleaving of active and passive behavior. The core language considered here ignores *ABS* features orthogonal to futures, including interface encapsulation and local synchronous calls. We refer to a report for a treatment of these issues [Din et al., 2012a].

As in *ABS*, we assume language support for abstract data types, 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 $\in$. (This notation is somewhat shorter than the notation actually used in our implementation.)

3 Examples

We illustrate the ABS runtime assertion checking and theorem proving of ABS programs in KeY 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 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 (OpenRead), closeR (CloseRead), openW (OpenWrite) and closeW (CloseWrite). 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.

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

  {db = new DataBase();}

  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 in Reader/Writer Example

Internally in the class, the attribute readers contains a set of clients currently with read access and writer contains the client with write access. Additionally, the attribute pr counts the number of pending calls to method read on object db. (A corresponding counter for writing is not needed since the call to method write on object db is called synchronously.) In order to ensure *fair* competition between readers and writers, invocations of openR and openW compete on equal terms for a guard writer = null. The set of readers is extended by execution of openR or openW, and the guards in both methods ensure that there is no writer. 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. The guard in openR will then be *false*, which means that new invocations openR will be delayed, and the write operations initiated by the writer will be delayed until the current reading activities are completed. 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. Thus even though readers may be non-empty while writer contains a client, the controller ensures that reading and writing *activity* cannot happen simultaneously on the database.

As in the Creol language [Johnsen and Owe, 2007], we use the convention that inside a method body the implicit parameter caller gives access to the identity of the caller. Thereby a caller need not send its identity explicitly; and the language guarantees that caller is not null. Conjunction is denoted &&. For readability reasons, we have taken the freedom to 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 are shown in Fig. 3.

```
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){ ProxyI lastProxy := this;
    if(#myClients < limit){ myClients := (myClients·cl);}
    else{if(nextProxy = null){ nextProxy := new Proxy(limit,s);}
      Fut<ProxyI> last := nextProxy!add(cl);
      lastProxy := last.get;}
    return lastProxy;}

  Void publish(Fut<News> fut){
    Int counter := 0; News ns := fut.get;
    while(counter < #myClients){
      ClientI client := myClients[counter];
      client!signal(ns); counter := counter + 1;}
    if(nextProxy = null){s!produce();}
    else{nextProxy!publish(fut);}}}
```

Figure 3: Implementation of class Service and Proxy in Publisher/Subscriber Example

The example takes advantage of the future concept by letting the service object delegate publishing of news updates to the proxies without waiting for the result of the news update. This is done by the sequence $fut := prod!detectNews(); proxy!publish(fut)$. Thus the service object is not blocking by waiting for news updates. Furthermore, the calls on *add* are blocking, however, this is harmless since the implementation of *add* may not deadlock and terminates efficiently. The other calls in the example are not blocking nor involving shared futures.

4 Observable Behaviour

In this section we describe a communication model for concurrent objects communicating by means of asynchronous message passing and futures. The model

is defined in terms of the *observable* communication between objects in the system. We consider how the execution of an object may be described by different *communication events* which reflect the observable interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [Broy and Stølen, 2001, 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, which is the one that *generates* the event. 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 o fetches 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 o creates a fresh object o'.

For a method call with future u, the ordering of events is described by the regular expression (using $\cdot$ for sequential composition of events)

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

for some fixed o, o', m, $\overline{e}$, e, and where _ denotes 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 created objects is fresh, and the communicating objects are non-null.

Invariants In interactive and non-terminating systems, it is difficult to specify and reason compositionally about object behaviour in terms of pre- and postconditions of the defined methods. Also, the highly non-deterministic behaviour of *ABS* objects due to processor release points complicates reasoning in terms of pre- and postconditions. Instead, pre- and postconditions to method definitions are in our setting used to establish a so-called *class invariant*. Class invariants express a relation between the internal state of class instances and observable communication. The internal state is given by the values of the class attributes.

> A class invariant must hold after initialization in all the instances of the class, be maintained by all methods, and hold at all processor release points (i.e., before **`await`** statements).

The five-event semantics reflects actions on asynchronous method calls, shared futures, and object creation. The semantics gives a clean separation of the activities of the different objects, which leads to disjointness of local histories. Thus, object behavior can be specified in terms of the observable interaction of the current object.

5 Runtime Assertion Checking

The *ABS* compiler front-end, 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 back-ends. 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.

We implement the history-explicit semantics in Maude as part of the ABS interpreter, by means of a global history reflecting all events that have occurred in the execution. We extend the *ABS* language with two annotations, **Require** and **Ensure**, given as part of method declarations:

```
[Require: pre_m] [Ensure: post_m && inv_m]
Unit m() {... assert inv_m; await b; ...}
Unit n() {... o!m(); ...}
```

where $\texttt{inv}_\texttt{m}$, $\texttt{pre}_\texttt{m}$ and $\texttt{post}_\texttt{m}$ are the class invariant, precondition and postcondition for the method *m*, respectively. The pre- and postconditions must then be respected by the method body, together with the invariant. Each method ensures that the invariant holds upon termination and when the method is suspended, assuming that the invariant holds initially and after suspensions. Preconditions indicate the values that a component is designed to process. Assertions impose constrains on variable values as a system passes through different execution states. Postconditions specify the functionality of a method by describing how its output values relate to its input values.

The underling implementation for the **Require** and **Ensure** annotations is shown in Fig. 4, where a Java method translating *ABS* method declarations into Maude. The properties defined in the **Require** and **Ensure** annotations are implemented as assertions around the method body.

The *invocation reaction events* are generated by callees once the methods start. The *future events* are generated by callees upon the execution of return statements which appear in the end of the method bodies. The order of extending the histories for method execution, inlining the **Require** and **Ensure** annotations as well as executing the method body can be presented in Fig. 5, where `invocrEv` and `futureEv` stand for the generation of *invocation reaction event* and *future event*, respectively; `Require` and `Ensure` locate the execution of the expressions defined in the method annotations. This shows that we can define a precondition expressing the property of `invocrEv` and a postcondition expressing the property of `futureEv`.

In this work, history functions are implemented in Maude as part of the *ABS* interpreter to extract relevant information from the generated history, such that history-based invariants are subjected to runtime assertion checking.

```
public void MethodImpl.generateMaude
 (PrintStream stream){
  PureExp require =
    CompilerUtils.getAnnotationValue(
    sig.getAnnotationList(), "Require");
  PureExp ensure =
    CompilerUtils.getAnnotationValue(
    sig.getAnnotationList(), "Ensure");
    stream.print("< "+...+": Method |
                  Param: "+" ... ");
    stream.print(",\n Code: ");
    if (require != null){
      stream.print("assert ");
      require.generateMaude(stream);
      stream.print("; ");}
  /* print out the method body */
  /* including the return statement */
    if (ensure != null) {
      stream.print("assert ");
      ensure.generateMaude(stream);}
    stream.print("> ");
}
```

Figure 4: The Java method which translates *ABS* method declarations with **Require**/**Ensure** annotations into Maude.

5.1 Specification and Verification of the Reader/Writer Example

For the RWController class, we may define a class invariant $I \triangleq I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5$:

$$\begin{array}{lcl} I_1 & \triangleq & \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \\ I_2 & \triangleq & \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} - \mathsf{null} \\ I_3 & \triangleq & \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \\ I_4 & \triangleq & \mathit{Writing}(\mathcal{H}) \neq 0 \Rightarrow \#\mathit{Writers}(\mathcal{H}) = 1 \\ & & \wedge\, \mathit{Readers}(\mathcal{H}) \subseteq \mathit{Writers}(\mathcal{H}) \\ I_5 & \triangleq & \mathit{Writing}(\mathcal{H}) = 0 \vee \mathit{Reading}(\mathcal{H}) = 0 \end{array}$$

This illustrates how the values of class attributes may be expressed in terms of observable communication. The invariant I_1 expresses that the set of readers retrieved from the history by function $\mathit{Readers}(h)$ is the same as the class attribute readers. The invariant I_2 expresses that if the set of writers retrieved from the history by function $\mathit{Writers}(h)$ 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 function $\mathit{Reading}(h)$ in I_3 computes the difference between the number of initiated calls to db!read and the corresponding get statement. The function

```
invocrEv Require methodBody futureEv
Ensure
```

Figure 5: The implementation order of **Require** and **Ensure** annotations, and history extension for method declaration.

Writing(h) is defined in a corresponding manner. The invariant I_4 states that when the database is being written, there should be one and only one writer. Besides, the set of readers may only contain the writer object. The invariant I_5 implies the desired mutual exclusion of reading and writing, i.e., no reading and writing *activity* happens simultaneously.

For the implementation details of the history functions, we define *Readers* : *Seq*[*Ev*] → *Set*[*Obj*]:

$$
\begin{aligned}
&\mathit{Readers}(\mathsf{Nil}) \triangleq \mathsf{Empty}\\
&\mathit{Readers}(h \cdot \langle \leftarrow \mathsf{this}, \mathit{fr}', \mathsf{openR}, \_\rangle) \triangleq\\
&\qquad \mathit{Readers}(h) + \mathit{irev}(h, \mathit{fr}').\mathit{caller}\\
&\mathit{Readers}(h \cdot \langle \leftarrow \mathsf{this}, \mathit{fr}', \mathsf{openW}, \_\rangle) \triangleq\\
&\qquad \mathit{Readers}(h) + \mathit{irev}(h, \mathit{fr}').\mathit{caller}\\
&\mathit{Readers}(h \cdot \langle \leftarrow \mathsf{this}, \mathit{fr}', \mathsf{closeR}, \_\rangle) \triangleq\\
&\qquad \mathit{Readers}(h) - \mathit{irev}(h, \mathit{fr}').\mathit{caller}\\
&\mathit{Readers}(h \cdot \langle \leftarrow \mathsf{this}, \mathit{fr}', \mathsf{closeW}, \_\rangle) \triangleq\\
&\qquad \mathit{Readers}(h) - \mathit{irev}(h, \mathit{fr}').\mathit{caller}\\
&\mathit{Readers}(h \cdot \boldsymbol{\mathit{others}}) \triangleq \mathit{Readers}(h)
\end{aligned}
$$

where ***others*** matches all events not matching any of the above cases. The function $\mathit{irev}(h, \mathit{fr}')$ extracts the invocation reaction event, containing the future fr', from the history h. The caller is added to the set of readers upon termination of openR or openW, and the caller is removed from the set upon termination of closeR or closeW. We furthermore assume a function *Writers*, defined over completions of openW and closeW in a corresponding manner. Next we define *Reading* : *Seq*[*Ev*] → *Nat* by:

$$
\begin{aligned}
\mathit{Reading}(h) \triangleq \#((h/\langle \mathsf{this} \rightarrow \mathsf{db}, \_, \mathsf{read}, \_\rangle).\mathit{future}\\
-(h/\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle).\mathit{future})
\end{aligned}
$$

where projection, $\_/\_ : \mathit{Seq}[T] \times \mathit{Set}[T] \rightarrow \mathit{Seq}[T]$ is defined inductively by $\mathsf{Nil}/s \triangleq \mathsf{Nil}$ and $(a \cdot x)/s \triangleq$ **if** $x \in s$ **then** $(a/s) \cdot x$ **else** a/s, for $a : \mathit{Seq}[T]$, $x : T$, and $s : \mathit{Set}[T]$, restricting a to the elements in s. And $(h/\rightarrow).\mathit{future}$ is the set of future identities from these invocation events. Thus the function *Reading*(h) computes the difference between the number of initiated calls to db!read and the corresponding get statements. The function *Writing(h)* follows the same pattern over calls to db!write.

Implementation The global history is implemented as part of the *ABS* interpreter and is not transparent in *ABS* programs, therefore we implement the history functions, *Readers*(h), *Writers*(h), *Reading*(h), and *Writing*(h), in the *ABS* interpreter and provide for each function a built-in predicate in the *ABS* language. Note that none of the predicates have the history as an explicit argument. For instance, the built-in predicate *getReaders*() returns the result of *Readers*(h) from the interpreter. Similarly, *getWriters*() corresponds to *Writers*(h), *numAccessing*("*read*") corresponds to *Reading*(h), and *numAccessing*("*write*") corresponds to *Writing*(h).

The concrete formulation of $I_1 - I_5$ as given to the runtime assertion checker is presented below:

```
I1: compareSet(getReaders(), readers)
I2: compareSet(getWriters(),
```

```
    Empty + writer − null)
I3: numAccessing("read") = pr
I4: numAccessing("write") ≠ 0 ⇒
    #getWriters() = 1 ∧
    isSubset(getReaders(),getWriters())
I5: numAccessing("read") = 0 ∨
    numAccessing("write") = 0
```

where `compareSet(`s_1`,` s_2`)` returns true if the set s_1 is equal to s_2, and `isSubset(`s_1`,` s_2`)` returns true if the set s_1 is a subset of the set s_2.

Another approach is to define a built-in function in *ABS* which gives the current local history from the interpreter. In this way, *ABS* programmers have the freedom to define their own history functions in *ABS* programs. However, the arguments of a method call can be a list of values of different types. This proposal is currently not considered due to the absence of a universal supertype in the *ABS* language (as the predefined type *Data* of the Creol language [Johnsen and Owe, 2007]).

5.2 Specification and Verification of the Publisher/Subscriber Example

In the publisher-subscriber example we consider object systems based on the classes Service and Proxy found in Fig. 3. We may state some general properties, like the following one: 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 service.

Since this property depends on pattern matching, we define an algebraic data type *Event* in *ABS*, and the constructors of the type are listed in Fig. 6 as the following:

```
InvocEv(callee,future,method,args)
InvocrEv(caller,future,method,args)
FutureEv(future,method,result)
FetchEv(future,result)
NewEv(callee,className,args)
```

Figure 6: *ABS* events.

This event type is used to define class invariants with simple pattern matching, to be verified at runtime for each related object. The generating object is redundant in the local invariants. Therefore, compared to the events defined in the *ABS* interpreter, explained in Sec. 4, we omit the generating objects from the events defined in *ABS*. Since there is no supertype 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` as shown in Fig. 7, letting all arguments in the events be of type `Any` except `method` and `className`, which are of type `String`. The constructors of type `Any`

```
data Any = O | F | AR | any ;
data Event =
   InvocEv(Any o, Any f, String m, Any a)
 | InvocrEv(Any o, Any f, String m, Any a)
 | FutureEv(Any f, String m, Any r)
 | FetchEv(Any f, Any r)
 | NewEv(Any o, String c, Any a) ;
```

Figure 7: Data types of events.

are `any`, a special constant used as a placeholder for any expression, and `O`, `F`, and `AR`, are artificial constants used as placeholders 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. $\texttt{O}_1$ and $\texttt{O}_2$ for object identities. Compared to dynamic logic specifications there is no universally quantified variables or auxiliary variables defined in the class invariants for runtime assertion checking. We therefore need these placeholders to express non-regular pattern matching, (when the same placeholder appears more than once).

Now we may derive the property above using the class invariants shown in Fig. 8, in which the class invariants for *Service* is $I_1 \wedge I_2$ and for *Proxy* is $I_3 \wedge I_4$. 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* returns *true* if the list of events is a subsequence of the local history. The search for a subsequence by the implementation of *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

```
I1:
 has(InvocEv(any,any,"add",any)) =>
 isSubseq(list[InvocEv(any,any,"add",AR),
      InvocrEv(any,any,"subscribe",AR)])
I2:
 has(InvocEv(any,any,"publish",any))=>
 isSubseq(list[InvocEv(any,any,"publish",
    F),InvocEv(any,F,"detectNews",any)])
I3:
 has(InvocEv(any,any,"signal",any)) =>
 isSubseq(list[InvocEv(O,any,"signal",AR),
 FetchEv(F,AR),InvocrEv(any,any,
 "publish",F),InvocrEv(any,any,"add",O)])
I4:
 has(InvocEv(any,any,"publish",any)) =>
 isSubseq(list[InvocEv(any,any,"publish",
 AR),InvocrEv(any,any,"publish",AR)])
```

Figure 8: Class invariants of the Publisher/Subscriber Example (1).

assertion checking. For instance, the invariant I_1 expresses that if the local history of the Service 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`. When we execute assertions, if I_1 holds for the current state, the Service object always receives a client before sending the client to the Proxy. Similar interpretation can be applied in I_2, I_3 and I_4.

By strengthening the class invariant of *Proxy*, we can also prove other properties such as: For each proxy, if *nextProxy* is null, the number of contained clients is less or equal to *limit*, otherwise equal to *limit*. The value of *nextProxy* can be reflected by the existence of an object creation event of Proxy as shown in Fig. 9. The symbol " $\sim$ " stands for negation.

```
I5: ~has(NewEv(_,"Proxy",_)) =>
     #myClients <= limit
I6: has(NewEv(_,"Proxy",_)) =>
     #myClients = limit
```

Figure 9: Class invariants of the Publisher/Subscriber Example (2).

6 Theorem Proving using KeY

In this section we describe our experiences with verification of some properties of the reader/writer and the publisher/subscriber examples.

6.1 Introduction to KeY

The KeY theorem prover [Beckert et al., 2007] is a deductive verification system. The standard system targets sequential Java programs while the system used in this case study is a variant that uses *ABS* programs instead of Java programs. The system features *ABS Dynamic Logic* which is a dynamic logic for *ABS*. We provide here only a quick overview on the logic for more details see [Ahrendt and Dylla, 2012].

ABSDL is basically a sorted first-order logic with modalities. Let $\mathtt{p}$ be an *ABS* program and ϕ an ABSDL formula, then the formula $[p]\phi$ is true if and only if the following holds: *if* $\mathtt{p}$ *terminates then in its final state* ϕ *holds.* Given an *ABS* method m with body mb and a class invariant I, the ABSDL formula to prove that method m preserves the class invariant by $I \rightarrow [mb]I$. The formula means that if method m is invoked in a state where the invariant holds initially then if m terminates in all reached final states the invariant holds again.

To prove that such a formula is valid we use a Gentzen-style sequent calculus. A *sequent* $\psi_1, \ldots, \psi_m \vdash \phi_1, \ldots, \phi_n$ has the same meaning as an implication where the formulas ψ_i on the left side of the sequent (antecedent) are conjunctively connected and the formulas ϕ_i on the right side (succedent) are connected disjunctively.

A proof in a sequent calculus is a tree which is constructed by a sequence of rule applications. The sequent calculus is based on the symbolic execu-

tion paradigm and emulates a symbolic interpreter for *ABS* programs. The conditional rule which *symbolically executes* a conditional statement looks as follows:

$$\frac{\Gamma, b \vdash [\texttt{p};\texttt{rest}]\phi, \Delta \qquad \Gamma, b \vdash [\texttt{q};\texttt{rest}]\phi, \Delta}{\Gamma \vdash [\texttt{if(b)\{p\}else\{q\}};\texttt{rest}]\phi, \Delta}$$

where Γ, Δ stand for (possibly empty) sets of formulas. Rules are applied from bottom to top by matching the sequent of the bottom part (the rule's conclusion) against the sequent of an open proof goal (leaf) of the proof. The conditional rule matches any sequent which contains an *ABS* program whose first active statement is a conditional statement. Application of the rule splits the proof into two branches. The left branch assumes that the guard of the conditional statement is true. Here, we have to show that after execution of the *then* branch of the conditional and the rest of the program, we are in a state in which formula ϕ holds. The right branch is concerned with the analogue case where the guard is assumed to be false.

6.2 Reasoning about concurrent and distributed *ABS* programs

We describe briefly how to formalize and reason about *ABS* programs in presence of concurrency and distribution. First, the history is modeled as an explicit list of events. The current history is maintained by a global program variable called $\mathcal{H}$. The program variable $\mathcal{H}$ cannot be directly assigned within an *ABS* program, but for instance the symbolic execution of an asynchronous method invocation updates the variable by appending an invocation event to the history.

One feature of our logic is that we do not need to model process queues or interleaving explicitly, but we can stay in a sequential setting. The concurrency model of *ABS* ensures co-operative multithreading for threads within the same object. This means we can syntactically determine interleaving points, i.e., places in the code where control is released. In our setting the only statements that release control are await statements. The rule for the await statement looks (slightly simplified) as follows:

$$\frac{\begin{array}{l}\Gamma \vdash CInv_C(\mathcal{H}, \bar{A}), \Delta \\ \Gamma \vdash \{U_{\mathcal{H},\bar{A}}\}(CInv_C(\mathcal{H}, \bar{A}) \rightarrow [\texttt{rest}]\phi), \Delta\end{array}}{\Gamma \vdash [\texttt{await r?};\texttt{rest}]\phi, \Delta}$$

In case of an **`await`** statement, the proof splits into two branches. The first branch ensures that the class invariant is satisfied before control is released, so that the process which takes over control can assume that the class invariant holds. The second branch is concerned with the continuation of the program execution once the future `r` is resolved and the current thread is rescheduled for execution. In such a state, all instance variables might have been changed and the history might have been extended by new events by other processes while the execution of our thread was suspended. This is technically represented by the update $U_{\mathcal{H},\bar{A}}$. For the purpose of this paper, updates can be seen as explicit substitutions which represents state changes. The update $U_{\mathcal{H},\bar{A}}$ sets all fields to fixed but unknown values and extends the history $\mathcal{H}$ by appending a sequence of unknown length and content. The formula behind the update, states that assuming the class invariant holds in the new state then we have to show that

after the execution of the remaining program `rest` the formula ϕ holds. One additional remark, the need to extend the history with an unknown sequence of events is one of the causes that makes verification tedious as it removes almost all knowledge about the system state and the only properties we can exploit are those encoded in the class invariant.

6.3 Formalizing and Verifying the Reader Writer Example

Formalization of the invariants and proof-obligations for the purpose of verification proves harder than is the case 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_2 from Section 5.1. Invariant I_2 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 our dynamic logic (slightly beautified) looks as follows:

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

where h is a 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_2 for an object `self`:

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

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* and *invocation reaction events* for

`openW` and `closeW`. This allows us to simplify the history at several places and to ease the proving process.

In order to prove the property of fairness and mutual exclusion of the reader writer example, we might need a stronger invariant: $OK : Seq[Ev] \to Bool$ which is defined inductively over the history:

$$
\begin{aligned}
&OK(\mathsf{Nil}) \triangleq \mathit{true}\\
&OK(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{openR}, \_\rangle) \triangleq OK(h)\\
&\qquad\qquad \wedge \#\mathit{Writers}(h) = 0 \quad (1)\\
&OK(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{openW}, \_\rangle) \triangleq OK(h)\\
&\qquad\qquad \wedge \#\mathit{Writers}(h) = 0 \quad (2)\\
&OK(h \cdot \langle \mathsf{this} \to \mathsf{db}, fr', \mathsf{write}, \_\rangle) \triangleq OK(h)\\
&\qquad \wedge \mathit{Reading}(h) = 0 \wedge \#\mathit{Writers}(h) = 1 \quad (3)\\
&OK(h \cdot \langle \mathsf{this} \to \mathsf{db}, fr', \mathsf{read}, \_\rangle) \triangleq OK(h)\\
&\qquad\qquad \wedge \mathit{Writing}(h) = 0 \quad (4)\\
&OK(h \cdot \boldsymbol{others}) \triangleq OK(h)
\end{aligned}
$$

Here, conditions (1) and (2) reflect the *fairness condition*: invocations of openR and openW compete on equal terms for the guard writer = null, which equals $\mathit{Writers}(\mathcal{H}) = \mathsf{Empty}$ by I_2. If writer is different from null, conditions (1) and (2) additionally ensure that no clients can be included in the readers set or be assigned to writer. Condition (3) captures the guard in write: when invoking db!write, there cannot be any pending calls to db!read. Correspondingly, condition (4) expresses that when invoking db!read, there is no incomplete writing operation.

6.4 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 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 for detecting the presence of bugs, while formal verification provides a guarantee that the program has the required property. In other words, although by using runtime assertion checking we gain confidence in the quality of programs, correctness of the software is still not fully guaranteed for all runs. Formal verification may instead show that a program is correct by proving that the code satisfies a 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, we extend the history by an unspecified sequence of events before continuing the execution after those release points. The only knowledge about the new extended history stems from the invariants.

The necessary specifications turned out to rely heavily on quantification and recursive defined functions and 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. 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, due to the combination of concurrency, compositionality, and object orientation. Moreover, the gap in reasoning complexity between sequential and distributed, object-oriented systems makes tool-based verification difficult in

practice. A survey of these challenges can be found in [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, and remote access to object fields are forbidden. By creating unique references for method calls, the *label* construct of Creol [Johnsen and Owe, 2007] resembles futures, as callers may postpone reading result values. 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 with other objects. A reasoning system for futures has been presented in [de Boer et al., 2007], using a combination of global and local invariants. Futures are treated as visible objects rather than reflected by events in histories. In contrast to our work, global reasoning is obtained by means of global invariants, and not by compositional rules. Thus the environment of a class must be known at verification time.

A compositional reasoning system for asynchronous methods in ABS with futures is introduced in [Din et al., 2012b]. Compositional reasoning is ensured as history invariants may be combined to form global system specifications. In this work, we implement the reasoning system [Din et al., 2012b] in two ways: 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 runtime assertion checker for JML is implemented in [Cheon and Leavens, 2002]. However, history-based specification is not expressible in JML. A prototype of the verification system [Ahrendt and Dylla, 2012] based on the two-event semantics (for method calls) [Dovland et al., 2005] has been implemented in KeY [Beckert et al., 2007], but requires more complex rules than the present semantics.

9 Conclusions and Future Work

In this work we implement a runtime assertion checker and extend the KeY theorem prover for testing and verifying *ABS* programs, respectively. For runtime assertion checking, the ABS interpreter is augmented by an explicit representation of the global history, recording all events that occur in an execution. And the *ABS* modeling language is extended with method annotations such that users can define software behavioral specification [Hatcliff et al., 2012] inline with the code. We provide the ability to specify both state- and history-based properties, which are checked during simulation.

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 et al., 2009] 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.

[Ahern and Yoshida, 2007] Ahern, A. and Yoshida, N. (2007). Formalising Java RMI with Explicit Code Mobility. Theoretical Computer Science, 389(3):341 – 410.

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

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

[Baker Jr. and Hewitt, 1977] Baker Jr., H. G. and Hewitt, C. (1977). The Incremental Garbage Collection of Processes. In *Proc. 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59. ACM.

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

[Broy and Stølen, 2001] Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer.

[Cheon and Leavens, 2002] Cheon, Y. and Leavens, G. T. (2002). A runtime assertion checker for the java modeling language (jml). In *Proc. International Conference of Software Engineering Research and Practice*, pages 322–328. CSREA Press.

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

[Dahl, 1987] Dahl, O.-J. (1987). Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA.

[Dahl, 1992] Dahl, O.-J. (1992). *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y.

[de Boer et al., 2007] de Boer, F. S., Clarke, D., and Johnsen, E. B. (2007). A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer.

[Din et al., 2012a] Din, C. C., Dovland, J., and Owe, O. (2012a). An approach to compositional reasoning about concurrent objects and futures. Research Report 415, Dept. of Informatics, University of Oslo. Available at `http://folk.uio.no/crystald/`.

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

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

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

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

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

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

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

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

[Jeffrey and Rathke, 2005] 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 and Owe, 2007] Johnsen, E. B. and Owe, O. (2007). An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling, 6(1):35–58.

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

[Yonezawa et al., 1986] Yonezawa, A., Briot, J.-P., and Shibayama, E. (1986). Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268.