Run-Time Verification of Coboxes

Frank S. de Boer[1,2], Stijn de Gouw[1,2], and Peter Y. H. Wong[3]

[1] CWI, Amsterdam, The Netherlands
[2] Leiden University, The Netherlands
[3] SDL Fredhopper, Amsterdam, The Netherlands

Abstract. Run-time assertion checking is one of the most useful techniques for detecting faults, and can be applied during any program execution context, including debugging, testing, and production. In this paper we show how to model the observable behavior of concurrently running object groups (coboxes) in SAGA (Software trace Analysis using Grammars and Attributes) which is a run-time checker that provides a smooth integration of the specification and the run-time checking of both data- and protocol-oriented properties of message sequences. We illustrate the effectiveness of our method by an industrial case study from the eCommerce software company Fredhopper.

1 Introduction

In [18] Java is extended with a concurrency model based on the notion of concurrently running object groups, so-called coboxes, which provide a powerful generalization of the concept of active objects. Coboxes can be dynamically created and objects within a cobox have only direct access to the fields of the other objects belonging to the same cobox. Since one of the main requirements of the design of coboxes is a smooth integration with object-oriented languages like Java, coboxes themselves do not have an identity, e.g., all communication between coboxes refer to the objects within coboxes. Communication between coboxes is based on asynchronous method calls with standard objects as targets. An asynchronous method call spawns a local thread within the cobox to which the targeted object belongs. Such a thread consists of the usual stack of internal method calls. Coboxes support multiple local threads which are executed in an interleaved manner. The local threads of a cobox are scheduled cooperatively, along the lines of the Creol modelling language described in [12]. This means, that at most one thread can be active in a cobox at a time, and that the active thread has to give up its control explicitly to allow other threads of the same cobox to become active.

In order to be able to understand and verify the overall behavior of a system in terms of its concurrently running coboxes suitable abstractions are absolutely essential. In this paper we first capture by means of a new formal semantics the relevant observable behavior of a cobox. More specifically, we show that for pure asynchronous systems of coboxes which only communicate via asynchronous method calls, e.g., no support is included for synchronization on return

values by futures [6], simple sequences of input/output messages which only refer to the targeted objects suffice for a compositional semantics. As such these sequences provide a powerful abstraction of the internal multithreaded flow of control within a cobox.

The main problem addressed in this paper is the run-time verification of coboxes as introduced recently in [18]. Run-time assertion checking is one of the most useful techniques for detecting faults, and can be applied during any program execution context, including debugging, testing, and production [4]. We provide a new compositional semantics for a specific class of coboxes. This semantics supports a formal definition of behavioral interfaces in terms of sequences of input/output messages and matches the abstraction level of coboxes in that messages only refer to objects. We show how to use attribute grammars extended with assertions to specifiy and verify at run-time properties of the messages sent between coboxes. To this end we extend the run-time assertion checking tool SAGA described in [7] which smoothly integrates both data- and protocol-oriented properties of message sequences. We illustrate the effectiveness of our method by an industrial case study from the eCommerce software company Fredhopper.

Plan of the Paper In the next section we first introduce an informal description of the case study which we use to illustrate our modelling and analysis techniques. Our modelling language and its formal semantics is described in section 3 and 4, respectively. Section 5 introduces the formalism used to describe behavioral interfaces. Tool-support is discussed in section 6 and a brief discussion of its use can be found in section 7. We conclude in the last section with related work and future research.

2 Case Study

The Fredhopper Access Server (FAS) is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture (See Figure 1).

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the *Replication Protocol*. The Replication Protocol is implemented by the *Replication System*. The Replication System consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the *schedule* of replication, as well as its content, while SyncClient receives data and configuration updates according to the schedule.

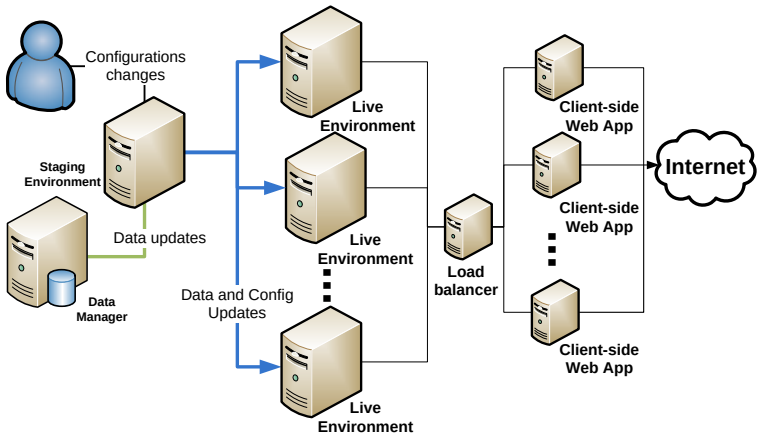

Fig. 1. An example FAS deployment

Replication Protocol

The SyncServer communicates to SyncClients by creating *Worker* objects. Workers serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering the read-/write access of the staging environment's underlying file system, the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 2 shows a UML sequence diagram concerning parts of the replication protocol with the interaction between a SyncClient, a ClientJob, a Worker, a SyncServer, a Coordinator and a Snapshot. The figure assumes that SyncClient has already established connection with a SyncServer and shows how a ClientJob from the SyncClient and a Worker from a SyncServer are instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a *replication session*. We now informally describe the interaction between the ClientJob and the Worker:

> The ClientJob initially connects to a Worker (`SyncServer.getConnection`, `ClientJob.acceptConnection`); the ClientJob then requests the next set of replication schedules from the Worker (`Worker.command`, `ClientJob.sendSchedule`); After that the Worker registers with the ClientJob the data to be replicated (`ClientJob.registerItems`, `Worker.replyRegisterItems`); Should the ClientJob accept the registration, the Worker proceeds sending to the ClientJob (meta information) of files to be replicated

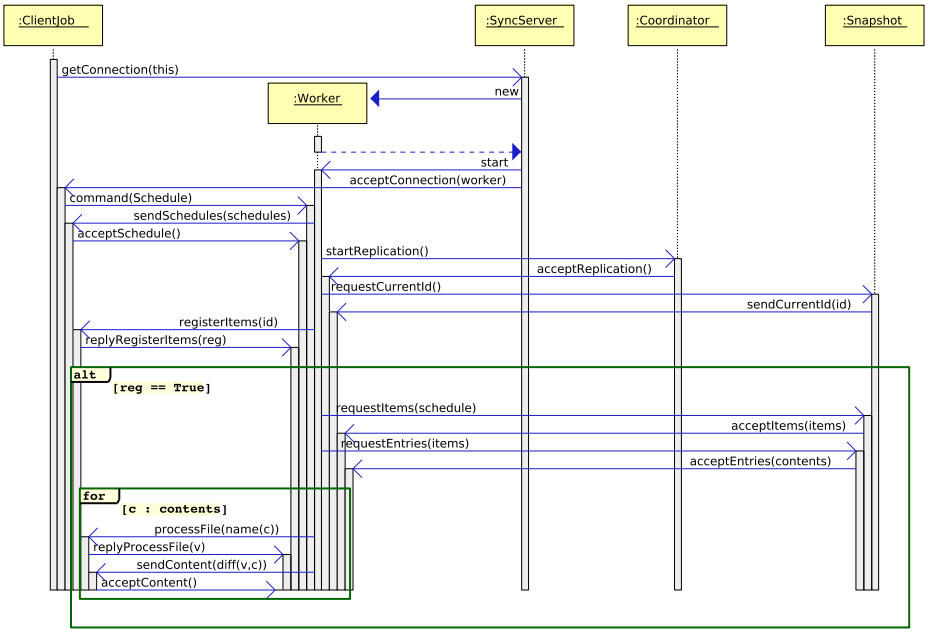

Fig. 2. Replication interaction

(`ClientJob.processFile`, `Worker.replyProcessFile`). For each of the files the ClientJob replies to the Worker indicating which part of the files need to be replicated, and with this information Worker sends relevant parts of the files to the ClientJob (`ClientJob.sendContent`, `Worker.acceptContent`).

3 The Modeling Language

The modeling language discussed in this paper is based on the ABS [11] which is an abstract, executable, object-oriented modeling language with a formal semantics, targeting distributed systems. ABS is designed with a layered architecture, at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) Java. ABS generalizes the concurrency model of Creol [12] from single concurrent objects to concurrent object groups (coboxes). As in [18] coboxes encapsulate synchronous, multi-threaded, shared state computation on a single processor. An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows to write concurrent programs in a much less error-prone way than in a thread-based model and makes ABS models suitable for static analysis. Differently from [18] in our dialect coboxes communicate only via pure asynchronous messages, and

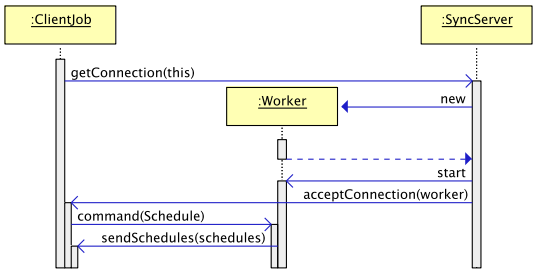

Fig. 3. Connecting to Worker and Acquiring Replication Schedules

as such form an actor-based model as initially introduced by [1] and further developed in [19].

In this subsection we describe the core constructs of our dialect of the ABS used in this paper in some detail. Specifically, we describe

- algebraic data types and functions;
- interfaces and classes;
- synchronous method calls and objects creation;
- asynchronous method calls and cobox creation;
- cooperative scheduling using **await** statements.

To illustrate synchronous and asycnhronous communications we look at the implementation of how a ClientJob connects to a Worker and receives the next set of replication schedules. This part of the protocol is illustrated in the UML sequence diagram in Figure 3.

Data types and Functions ABS supports *algebraic data types* (ADT) to model data in a software system. ADTs abstract away from implementation details such as hardware environment, file content, or operating system specifics. For example in the Replication System, the following ADT `Content` models the file system of environments.

```
data Content = File(Int content) | Dir(Map<String,Content>);
```

ABS supports first-order functional programming with ADT. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features. For example, the following function `isFile` checks if the given `Content` value records a file.

```
def Bool isFile(Content c) = case { File(_) => True; _ => False; };
```

Interfaces ABS has a nominal type system with interface-based subtyping. Interfaces define types for objects. They have a name, and define a set of method signatures, that is, the names and types of callable methods. The following shows interface `Worker` that models a Worker.

```
interface Worker {
  Unit execute();
  Unit command(Command c);
  Unit acceptCoordinator(Coordinator coord);
  Unit sendCurrentId(Int id);
  Unit replyRegisterItems(Bool register);
  Unit acceptItems(Set<Item> items);
  Unit acceptEntries(Set<Map<String,Content>> contents); }
```

Classes ABS also supports class-based, object-oriented programming with standard imperative constructs. Classes define the implementation of objects. In contrast to Java, for example, classes do *not* define a type. Classes can implement arbitrarily many interfaces. These interfaces define the type of instances of that class. A class has to implement all methods of all its implementing interfaces. Instead of constructors, classes in ABS have *class parameters*, which are instance fields. In addition, a class may further define additional instance fields. The following class `WorkerImpl` implements `Worker`:

```
class WorkerImpl(ClientJob job, SyncServer server, Coordinator coord)
implements Worker {
  Maybe<Command> cmd = Just(ListSchedule);
  Unit execute() {..}
  Unit command(Command c) {..}
  Unit acceptCoordinator(Coordinator coord) {..}
  Unit sendCurrentId(Int id) {..}
  Unit replyRegisterItems(Bool register) {..}
  Unit acceptItems(Set<Item> items) {..}
  Unit acceptEntries(Set<Map<String,Content>> contents) {..}}
```

It defines class parameters `job` of `ClientJob` and `server` of `SyncServer`. Here `ClientJob` models a ClientJob and `SyncServer` models a SyncServer. It further defines instance fields `cmd` and `coord`, where `cmd` is initialized with some default value.

Thread-based computation Basic statements describing the flow of control of a single thread include the usual (synchronous) method invocations, object creation, and field and variable reads and assignments. These statements can be composed by the standard control structures (sequential composition, conditional and iteration constructs). The following shows the part of class `ClientJobImpl` that a ClientJob connecting to a Worker and acquiring the next schedules:

```
class ClientJobImpl(SyncServer server) implements ClientJob {
  Unit sendSchedules(Set<Schedule> ss) { .. }
  Unit executeJob() { .. }
  Unit acceptConnection(Worker w) {
```

```
    if (w != null) { .. this.scheduleJob(); }}
  Unit scheduleJobs() {
    Scheduler sr = new SchedulerImpl(..);
    sr.schedule(); }}
```

The method `acceptConnection` invokes synchronously the (private) method `scheduleJob`, which in turn creates an object of `SchedulerImpl` and invokes its method `schedule`.

Coboxes The concurrency model of ABS is based on the concept of *Coboxes*. A typical ABS system consists of multiple, concurrently running coboxes at run-time. Coboxes can be regarded as autonomous run-time components that are executed concurrently, share no state and communicate via method calls. A new object cobox is created by using the **new cog** expression. It takes as argument a class name and optional parameters and returns a reference to the initial object of the new cobox. Communication between coboxes may solely be done via *asynchronous method calls*. The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot. The following expands the fragment of `ClientJobImpl` shown earlier to illustrate cobox creation and asynchronous communications.

```
class ClientJobImpl(SyncServer server, SyncClient client, Schedule s)
implements ClientJob {
  Set<Schedule> schedules = EmptySet;
  Unit executeJob() { server!getConnection(this); }
  Unit acceptConnection(Worker w) { .. }
  Unit sendSchedules(Set<Schedule> ss) { .. }
  Unit scheduleJobs() { .. }}

class SyncServerImpl(Coordinator coord) implements SyncServer {
  Unit getConnection(ClientJob job) {
    Bool shutdown = this.isShutdownRequested();
    if (shutdown) {
     job!acceptConnection(null);
    } else {
      Worker w = new cog WorkerImpl(job, this, coord);
      job!acceptConnection(w); }}}
```

The classes `SyncServerImpl` implements the SyncServer and `ClientJobImpl` implements a ClientJob. `ClientJobImpl` has a class parameter `server` that holds the reference to the SyncServer that is assigned to a different cobox. The method `executeJob` invokes SyncServer's method `getConnection` asynchronously to connect with a Worker. In the implementation of `SyncServer`, a new object cobox is created with the `WorkerImpl` object being the initial object in that cobox.

Cooperative scheduling Each asynchronous method call results in a *task* in the cobox of the target object. Tasks are scheduled *cooperatively* within the scope of a object cobox. Cooperative scheduling means that switching between tasks of the same object cobox happens only at specific *scheduling points* during program execution and that at no point two tasks in the same cobox are active at the same time. Using the **await** statement, one can create a *conditional* scheduling point, where the running task is suspended until a Boolean condition over the object state becomes true. The following shows the implementation of `ClientJobImpl` after connecting with a `Worker`.

```
class ClientJobImpl(SyncServer server, SyncClient client, Schedule s)
implements ClientJob {
  Set<Schedule> schedules = EmptySet;
  Unit sendSchedules(Set<Schedule> ss) { schedules = ss; }
  Unit acceptConnection(Worker w) {
    if (w != null) {
      w!command(Schedule(s));
      await schedules != EmptySet;
      this.scheduleJobs();}}..}

class WorkerImpl(ClientJob job, SyncServer server) implements Worker {
  Unit command(Command c) { .. job!sendSchedules(schedules); }}
```

The method `acceptConnection` invokes method `command` on the worker and suspends using the statement **await** `schedules != EmptySet` to wait for the next set of schedules arrives. The next set of schedules is set by invoking the method `sendSchedules` on the ClientJob.

4 Semantics

In this section we describe the formal semantics of systems of coboxes compositionally in terms of its object coboxes. The behavior of a cobox itself is described compositionally in terms of its threads. In this section we abstract from the functional part of the modeling language. We further abstract from variable declarations and typing information, and simply assume given a set of variables $x, y, \ldots$. We distinguish between simple and instance variables. The set of simple variables is assumed to include the special variable "this". Simple variables are used as formal parameters of method definitions.

Throughout this section we assume a given program which specifies a set of classes and a (single) inheritance relation. We start with the following basic semantic notions. For each class C we assume given a set of O_C, with typical element o, of (abstract) objects which belong to class C at run-time. A *heap* h is formally given as a set of (uniquely) labelled object states $o : s$, where s assigns values to the instance variables of the object o. We denote $s(x)$, for $o : s \in h$, by $h(o.x)$. By s_{init} we denote the object state which results from the initialization of the instance variables of a newly created object. Further, by $h[o.x = v]$ we denote

the heap update resulting from the assignment of the value v to the instance variable x of the object o. Next we introduce a *thread configuration* as a pair $\langle t, h\rangle$ consisting of a thread t. and heap h. A thread itself is a stack of closures of the form (S, τ), where S is a statement and τ is a local environment which assigns values to simple variables. By $\tau[x = v]$ we denote the update of the local environment τ resulting from the assignment of the value v to the variable x. We denote by $V(e)(\tau, h)$ the value of a side-effect free expression e in the local environment τ and global heap h. In particular we have that $V(x)(s, h) = s(x)$, for a simple variable x, and $V(x)(\tau, h) = h(\tau(\text{this}).x)$, for an instance variable x.

Thread Semantics A transition

$$\langle t, h\rangle \longrightarrow \langle t', h'\rangle$$

between thread configurations $\langle t, h\rangle$ and $\langle t', h'\rangle$ indicates

- the execution of an assignment $x = e$ or
- the evaluation of a boolean condition b of an if-then-else or while statement,
- or the execution of a synchronous call.

A *labelled* transition

$$\langle t, h\rangle \xrightarrow{l} \langle t', h'\rangle$$

indicates for

$l = \text{await}$**:** the successfull execution of an await statement,
$l = o!m(\bar{v})$**:** an asynchronous call of the method m of the object o with actual parameters $\bar{v}$.

In the following structural operational semantics for the execution of single threads (we omit the transitions for sequential composition, if-then-else and while statement since they are standard) $(S, s) \cdot t$ denotes the result of pushing the closure (S, s) unto the stack t.

Assignment simple variables

$$\langle (x = e; S, \tau) \cdot t, h\rangle \longrightarrow \langle (S, \tau']) \cdot t, h\rangle$$

where $\tau' = \tau[x = V(e)(\tau, h)]$.

Assignment instance variables

$$\langle (x = e; S, \tau) \cdot t, h\rangle \longrightarrow \langle (S, \tau) \cdot t, h']\rangle$$

where $h' = h[\tau(\text{this}).x = V(e)(\tau, h)]$.

Await

$$\langle (\text{await } b; S, \tau) \cdot t, h\rangle \xrightarrow{\text{await}} \langle (S, \tau) \cdot t, h\rangle$$

where $V(b)(\tau, h) = \text{true}$.

Asynchronous method call

$$\langle (x!m(\bar{e}); S, \tau) \cdot t, h \rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle (S, \tau) \cdot t, h \rangle$$

where $o = V(x)(s, h)$, $\bar{e} = e_1, \ldots, e_n$, $\bar{v} = v_1, \ldots, v_n$, and $v_i = V(e_i)(s, h)$, for $i = 1, \ldots, n$.

Synchronous method call

$$\langle (y = x.m(\bar{e}); S, \tau) \cdot t, h \rangle \longrightarrow \langle (S', \tau') \cdot (y = r; S, \tau) \cdot t, h \rangle$$

where, assuming that $V(x)(s, h) \in O_C$, $m(\bar{x})\{S'\}$ is the corresponding method definition in class C. Further, $\tau'(\text{this}) = V(x)(\tau, h)$ and, for $i = 1, \ldots, n$, $\tau'(x_i) = V(e_i)(\tau, h)$, ($\bar{e} = e_1, \ldots, e_n$ and $\bar{x} = x_1, \ldots, x_n$,) We implicitly assume here that τ' initializes the local variables of m, i.e., those simple variables which are not among the formal parameters $\bar{x}$. Upon return the fresh simple variable r (which is assumed not to appear in the given program) will store the return value (see the transition below for returning a value).

Class instantiation

$$\langle (y = \text{new } C(\bar{e}); S, \tau) \cdot t, h \rangle \longrightarrow \langle (y = r.C(\bar{e}); S, \tau') \cdot t, h \cup \{o' : s_{init}\} \rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity, where C is the type of the variable y. The fresh variable r is used to store temporarily the identity of the new object. We implicitly assume that the constructor method returns the identity of the newly created object (by the statement "return this").

Cobox instantiation

$$\langle (y = \text{new cog } C(\bar{e}); S, \tau) \cdot t, h \rangle \longrightarrow \langle (y = r; y!C(\bar{e}); S, \tau') \cdot t, h \rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity, (C is the type of the variable y). As above, the fresh variable r is used to store temporarily the identity of the new object (here it allows to circumvent a case distinction on whether y is a simple or an instance variable). Note that the main difference with class instantiation is that the newly created object is *not* added to the heap h and the constructor method is called asynchronously.

In contrast to [11] and [18] we allow for very flexible scheduling policies (no assumptions are made about scheduling policies at all, even for constructors, besides the fact that **await** statements are respected), it is possible that the constructor method is executed at a later stage than a normal method called on the newly created object. If this is not desired, the user can synchronize explicitly using **await**.

Return

$$\langle(\text{return } e; S, \tau) \cdot (S', \tau') \cdot t, h\rangle \longrightarrow \langle(S', \tau'[r = v]) \cdot t, h\rangle$$

where $v = V(e)(\tau, h)$. The fresh variable r here is used to store temporarily the return value.

In the above transitions for the creation of a class instance or a new cobox we assume a thread-local mechanism for the selection of a fresh object identity which avoids name clashes between the activated threads, the technical details of which are straightforward and therefore ommitted.

Semantics of coboxes A cobox is a pair $\langle T, h\rangle$ consisting of a set T of threads and a heap h. An object o belongs to a cobox $\langle T, h\rangle$ if and only if it has a state in h, that is, $o : s \in h$, for some object state s.

Internal computation step
An unlabelled computation step of a thread is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \longrightarrow \langle t', h'\rangle}{\langle\{t\} \cup T, h\rangle \longrightarrow \langle\{t'\} \cup T, h'\rangle}$$

External call
A computation step labelled by an asynchronous method call is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle t', h'\rangle}{\langle\{t\} \cup T, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle\{t'\} \cup T, h'\rangle}$$

Synchronization
The execution of an await statement by a thread within a given cobox is formally captured by the rule

$$\frac{\langle t, h\rangle \overset{\text{await}}{\longrightarrow} \langle t', h'\rangle}{\langle\{t\} \cup T, h\rangle \longrightarrow \langle\{t'\} \cup T, h'\rangle}$$

provided all threads in T executing an await statement, that is, the top of each thread in T consists of a closure of the form $(\text{await } b; S, \tau)$ (we implicitly assume that terminated threads are removed). Note that thus the await statement enforces a barrier synchronization of all the threads of a cobox. This synchronization ensures that at most one thread in a cobox is executing.

Input-enabledness
We further have the following transition which describes the *reception* of an asynchronous method call to an object o which belongs to the cobox $\langle T, h\rangle$:

$$\langle T, h\rangle \overset{o?m(\bar{v})}{\longrightarrow} \langle T \cup \{t\}, h\rangle$$

where, , assuming that $o \in O_C$, $m(\bar{x})\{S\}$ is the corresponding method definition in class C. Further, t consists of the closure $\langle \text{await true}; S, \tau \rangle$, and τ assigns the actual parameters $\bar{v}$ to the formal parameters $\bar{x}$ of m (as above, the object identity o is assigned to the implicit formal parameter "this") and initializes all local variables of m.

The added await statement enforces synchronization between the other threads. Since coboxes are *input-enabled* this transition thus models *an assumption* about the environment. This assumption is validated in the context of coboxes as described next.

Semantics of systems of coboxes Finally, a system configuration is simply a set G of coboxes. For technical convenience we assume that all system configurations contain an infinite set of *latent* coboxes $\langle \emptyset, \{o : s_{init}\} \rangle$ which have not yet been activated. The fresh object generated by the creation of a new cobox, as described above in the thread semantics (transition 4), at this level is assumed to correspond to a latent cobox.

Interleaving
An internal computation step of a cobox is extended to a corresponding transition of the global system as follows.

$$\frac{g \longrightarrow g'}{\{g\} \cup G \longrightarrow \{g'\} \cup G}$$

Message passing
Communication between two coboxes is formalized by

$$\frac{g_1 \stackrel{o?m(\bar{v})}{\longrightarrow} g_1' \quad g_2 \stackrel{o!m(\bar{v})}{\longrightarrow} g_2'}{\{g_1, g_2\} \cup G \longrightarrow \{g_1', g_2'\} \cup G}$$

Here it is worthwhile to observe that for an asynchronous call $o!m(\bar{v})$ to an object o belonging to the *same* cobox there does not exist a matching reception $o?m(\bar{v})$ by a *different* cobox because coboxes have no shared objects.

Trace Semantics A trace is a finite sequence of input and output messages, e.g., $o?m(\bar{v})$ and $o!m(\bar{v})$, respectively. For each coboxes g we define its trace semantics $T(g)$ by

$$\{\langle \theta, g' \rangle \mid \ g \stackrel{\theta}{\longrightarrow} g'\}$$

where $\stackrel{\theta}{\longrightarrow}$ denotes the reflexive, transitive closure of the above transition relation between coboxes, collecting the input/output messages. Note that the trace θ by which we can obtain from g a cobox g' does *not* provide information about object creation or information about which objects belong to the same cobox. In fact, information about which objects have been created can be inferred from the trace θ. Further, in general a cobox does not "know" which objects belong to the same cobox.

The following compositionality theorem is based on a notion of *compatible* traces which roughly requires for every input message a corresponding output message, and vice versa. We define this notion formally in terms of the following rewrite rule for sets of traces

$$\{o?m(\bar{v}) \cdot \theta, o!m(\bar{v}) \cdot \theta'\} \cup \Theta \Rightarrow \{\theta, \theta'\} \cup \Theta$$

This rule identifies two traces in the given set which have two matching initial messages which are removed from these traces in the resulting set. Note that this identification is non-deterministic, i.e., for a given trace there may be several traces with a matching initial message. A set of traces Θ is compatible, denoted by $Compat(\Theta)$, if we can derive the singleton set $\{\epsilon\}$ (ϵ denotes the empty trace). Formally, $Compat(\Theta)$ if and only if $\Theta \Rightarrow^* \{\epsilon\}$, where $\Rightarrow^*$ denotes the reflexive, transitive closure of $\Rightarrow$.

Theorem 1. *Let $\rightarrow^*$ denote the reflexive, transitive closure of the above transition relation between system configurations. We have*

$$G \longrightarrow^* G'$$

if and only if $G = \{g_i \mid i \in I\}$ and $G' = \{g'_i \mid i \in I\}$, for some index set I such that for every $i \in I$ there exists $\langle \theta_i, g'_i \rangle \in T(g_i)$, with $Compat(\{\theta_i \mid i \in I\})$.

This theorem states that the overall system behavior can be described in terms of the above trace semantics of the individual coboxes. This means that for compositionality no further information is required. Next we show in the following section how to specify properties of the externally observable behavior of a cobox, as defined by its traces of input/output messages.

5 Behavioral Interfaces for Coboxes

In this section we introduce *attribute grammars* extended with assertions to specify and verify properties of the trace semantics as defined in the previous section. In contrast to classes or interfaces, coboxes are run-time entities which do not have a single fixed interface[4]. Below we first discuss how we can still refer statically, in the program text, to these run-time entities by means of so-called communication views.

5.1 Communication Views

To be able to refer to cobox in syntactical constructs (such as specifications), we introduce the following (optional) annotation of cobox instantiations:

$$S ::= y = \text{new cog [Name] } C(\bar{e})$$

[4] We consider interfaces here to be a list of all signatures of the methods supported by some object in the cobox

The semantics of the language remain unchanged. Note that the same name can be shared among several coboxes (i.e. is in general not unique) since different cobox creation statements can specify the same name.

Coboxes do not have a fixed interface, as the methods which can be invoked on an object in a cobox (and consequently appear in traces) are not fixed statically. In particular, during execution objects of any type can be added to a cobox, which clearly affects the possible traces of the cobox. Additionally, for practical reasons it is often convenient to focus on a particular subset of methods, leaving out methods irrelevant for specification purposes. This is especially useful for incomplete specifications. To solve both these problems, we introduce *communication views*. A communication view can be thought of as an interface for a named cobox. Figure 4 shows an example communication view associated to all coboxes named WorkerGroup. Formally a communication view is a par-

```
view WorkerView grammar Worker.g specifies WorkerGroup {
  send Coordinator.startReplication(Worker w) st,
  send ClientJob.registerItems(Worker w, Int id) pr,
  receive Worker.sendCurrendId(Int id) id,
  receive Worker.replyRegisterItems(Bool reg) ar,
  receive Worker.acceptItems(Set<Item> items) is,
  receive Worker.acceptEntries(Set<Map<String, Content>> contents) es
}
```

Fig. 4. Communication View

tial mapping from messages to abstract event names. A communication view thus simply introduces names tailored for specification purposes (see the next subsection about grammars for more details on how this name is used). Partiality allows the user to select only those asynchronous methods relevant for specification purposes. Any method not listed in the view will be irrelevant in the specification of WorkerGroups. The `send` keyword selects calls from objects in the WorkerGroup to methods of objects in another cobox, and corresponds to transitions labelled by $o!m(\bar{v})$ in the operational semantics. In other words: methods required by an object in the WorkerGroup. Conversely, the keyword `receive` selects calls from another cobox to an object in a WorkerGroup, which corresponds to transitions labelled by $o?m(\bar{v})$ in the semantics. It is possible that methods listed in the view actually can never be called in practice (and therefore won't appear in the local trace of a cobox). In the above view, this happens if in a WorkerGroup there is no object of the class `Worker`.

5.2 Grammars

In this subsection we describe how properties of the set of allowed traces of a cobox can be defined specified in a convenient, high-level and declarative manner. We illustrate our approach by partially specifying the behavior depicted by the UML sequence diagram in Figure 2. Informally the property we focus on is:

> The Worker first notifies the Coordinator its intention to commence a replication session, the Worker would then receive the last transaction id identifying the version of the data to be replicated, the Worker sends this id to the ClientJob to see if the client is required to update its data up to the specified version. The Worker then expects an answer. Only if the answer is positive can the Worker retrieves replication items from the snapshot, moreover, the number of files sets to be replicated to the ClientJob must correspond to the number of replication items retrieved.

Grammars provide a convenient way to define the protocol behavior of the allowed traces. The terminals of the grammar are the message names given in a communication view. The formalization of the above property uses the communication view depicted in Figure 4. The productions of the grammar underlying the attribute grammar in Figure 5 specify the legal orderings of these messages named in the view. For example, the productions

$$S ::= \epsilon \mid st \;\; T$$
$$T ::= \epsilon \mid id \;\; U$$

specify that the message 'id' is preceded by the message 'st'.
While grammars provide a convenient way to specify the *protocol structure* of the valid traces, they do not take data such as parameters and return values of method calls and returns into account. Thus the question arises how to specify the *data-flow* of valid traces. To that end, we extend the grammar with attributes and assertions over these attributes. Each terminal symbol has *built-in* attributes corresponding with the observables of the trace semantics defined in the previous section. The built-in attributes consist of the parameter names for referring to the object identities of the actual parameters, and `callee` for referencing the identity of the callee. Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. In each production, the value of the attributes of the non-terminals appearing on the right-hand side of the production is defined.[5] For example, in the following production, the attribute 'w' for the non-terminal 'T' is defined.

$$S ::= \epsilon \mid st \;\; T \; (T\texttt{.w} = st\texttt{.w;})$$

Attribute definitions are surrounded by '(' and ')'. However the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the trace. We extend the attribute grammar with assertions to specify properties of attributes. For example, the assertion in the second production of

$$T ::= \epsilon \mid id \;\; U \; (U\texttt{.w} = T\texttt{.w; } U\texttt{.i} = id\texttt{.id;})$$
$$U ::= \epsilon \mid pr \; \{\textbf{assert } U\texttt{.w} == pr\texttt{.w \&\& } U\texttt{.i} == pr\texttt{.id;}\} \; V$$

expresses that the 'id' passed as a parameter to the method 'registerItems' (represented in the grammar by the terminal $pr\texttt{.id;}$) must be the same as the

[5] In the literature, such attributes are called inherited attributes.

one previously passed into 'sendCurrentId' (terminal id.id). Assertions are surrounded by '{' and '}' to distinguish them visually from attribute definitions.

The full attribute grammar Figure 5 formalizes the informal property stated in the beginning of this subsection. The grammar specifies that for each Worker object, in its own object cobox, the Coordinator must be notified of the start of the replication by invoking its method `startReplication` (st). Only then can the Worker receive (from an unspecified cobox) the identifier of the current version of the data to be replicated (id). Next the Worker invokes the method `registerItems` on the corresponding ClientJob about this version of the data (pr). The grammar here asserts that the identifier is indeed the same as that received via the method call `sendCurrendId`. The Worker then expects to receive a method call `replyRegisterItems` indicating if the replication should proceed, the Worker then can recieve method call `acceptItems` for the data items to be replicated. The grammar here asserts that this can only happen if the previous call indicated the replication should proceed. The Worker then can receive method call `acceptEntries` for the set of Directories, each identified by a data item. Since each data item refers to a directory, the grammar here asserts the number of items is the same as the number of directories.

$$
\begin{array}{l}
S \ ::= \epsilon \mid st \ \ T \ (T\texttt{.w} = st\texttt{.w;}) \\
T \ ::= \epsilon \mid id \ \ U \ (U\texttt{.w} = T\texttt{.w;}\ U\texttt{.i} = id\texttt{.id;}) \\
U \ ::= \epsilon \mid pr \ \{\textbf{assert}\ U\texttt{.w} == pr\texttt{.w \&\& } U\texttt{.i} == pr\texttt{.id;}\}\ V \\
V \ ::= \epsilon \mid ar \ \ W \ (W\texttt{.b} = ar\texttt{.reg;}) \\
W ::= \epsilon \mid is \ \{\textbf{assert}\ W\texttt{.b;}\}\ X \ (X\texttt{.s} = \texttt{size(}is\texttt{.items);}) \\
X \ ::= \epsilon \mid es \ \{\textbf{assert}\ X\texttt{.s} == \texttt{size(}es\texttt{.contents);}\}
\end{array}
$$

Fig. 5. Attribute Grammars

```
view ScheduleView grammar Schedule.g specifies WorkerGroup {
  receive Worker.command(Command c) cm,
  send ClientJob.sendSchedules(Set<Schedule> ss) sn,
  send SyncServer.requestListSchedules(Worker w) lt,
  send SyncServer.requestSchedule(Worker w, String name) gt,
  send Coordinator.requestStartReplication(Worker w) st
}
```

Fig. 6. Communication View for Scheduling

To further illustrate the above concepts, we consider an additional behavioral interface for the WorkerGroup cobox. To allow users to make changes the replication schedules during the run-time of FAS, every ClientJob would request the next set of replication schedules and send them to SyncClient for schedul-

$$
\begin{array}{l}
S ::= \epsilon \mid cm \;\; T \; (T\texttt{.c} = cm\texttt{.c;}) \\
T ::= \epsilon \mid gt \;\; \{\textbf{assert } T\texttt{.c != ListSchedule \&\&} \\
\qquad\qquad\quad gt\texttt{.n == name(}T\texttt{.c);}\} \; U \; (U\texttt{.c} = T\texttt{.c;}) \\
\qquad\quad \mid lt \;\; \{\textbf{assert } T\texttt{.c == ListSchedule;}\} \; U \; (U\texttt{.c} = T\texttt{.c;}) \\
U ::= \epsilon \mid sn \;\; \{\textbf{assert } sn\texttt{.ss != EmptySet;}\} \; V \; (V\texttt{.c} = U\texttt{.c;}) \\
V ::= \epsilon \mid st \;\; \{\textbf{assert } V\texttt{.c != ListSchedule;}\}
\end{array}
$$

Fig. 7. Attribute Grammar for Scheduling

ing. Here is an informal description of the property, where Figure 6 presents the communication view capturing the relevant messages and Figure 7 presents the grammar that formalizes the property:

> A ClientJob may request for either all replication schedules or a single schedule. The ClientJob does this by sending a command to the Worker (cm). If the command is of the value `ListSchedule`, the Worker is to acquire all schedules from the SyncServer (lt) and return them to the ClientJob (sn). Otherwise, the Worker is to acquire only the specified schedule (gt) and return it to the ClientJob (sn). If the ClientJob asks for all schedules, it must not proceed further with the replication session and terminate (st).

In summary, communication views provide an interface of a named cobox. The behavior of such an interface is specified by means of an attribute grammar extended with assertions. This grammar represents the legal traces of the named cobox as words of the language generated by the grammar, which gives rise to a natural notion of the satisfaction relation between programs and specifications. Properties of the control-flow and data-flow are integrated in a single formalism: the grammar productions specify the valid orderings of the messages (the control-flow of the valid traces), whereas assertions specify the data-flow.

We conclude this section with the semantic definition of a program extended with communication views and attribute grammars. To this end, we assume for any set $\{g_i \mid \; i \in I\}$ of coboxes of a program annotated with cobox names that the communication view and corresponding attribute grammar of a cobox g_i is given by V_i and A_i, respectively. Note that this requires that we can identify the name of a group at run-time (the corresponding extension of the semantics is straightforward and therefore omitted). Further, for any group g we denote by $\mathit{Trace}(g)$ the set of traces of $T(g)$ (see the Trace Semantics paragraph directly above Theorem 1 for a definition of $T(g)$). For any attribute grammar we denote by $\mathit{Trace}(A)$ the set of traces generated by the grammar. Finally, by $\theta \downarrow_V$ we denote the projection of the trace θ unto the messages specified by the communication view V.

Definition 1. *Let P be a program annotated with cobox names and S be a specification consisting of a communication view and corresponding attribute grammar for each named cobox. Then P satisfies S if and only if for all system config-*

urations $G = \{g_i \mid i \in I\}$ *of* P *and sets of traces* $\Theta = \{\theta_i \mid \theta_i \in \mathit{Trace}(g_i),\ i \in I\}$ *such that* $\mathit{Compat}(\Theta)$ *we have* $\theta_i \downarrow_{V_i} \in \mathit{Trace}(A_i)$*, for* $i \in I$*.*

6 Implementation

In this section we discuss the architecture of our run-time checker SAGA, crucial design decisions and its performance. SAGA is implemented as a run-time checker for ABS models. ABS is basically an extension of the modeling language considered in this paper. It is tool-supported by various analysis tools [21] and automated code generation has been implemented to various lower-level languages including Java, Maude and Scala. SAGA tests whether an actual execution of a given ABS model satisfies its specification given by attribute grammars, and stops the running program in case of a violation to prevent unsafe behavior. It is implemented as a meta-program in Rascal [13]. Rascal is a meta-programming language featuring powerful techniques for parsing and source code analysis, transformation and generation.

Design The design of SAGA was guided by several requirements.

1. *All* back-ends (even future ones) which generate code from ABS models to lower-level target languages should be supported, without having to update SAGA when any of the back-ends is updated (for example, to generate more efficient code). Consequently we need a parser-generator which generates ABS code, and therefore cannot use existing parser generators.
2. The overhead induced by SAGA must be kept to a minimum. In particular, whenever the trace of a cobox is updated with a new message, SAGA should be able to decide *in constant time* whether the new trace still satisfies the specification (the attribute grammar). This is determined by parsing the trace (then considered as a sequence of tokens) in a parser for the attribute grammar.
3. Because of the intrinsic complexity of developing efficient and user-friendly parser generators, we require that the implementation of the parser-generator should be decoupled from the rest of the implementation of SAGA.

These requirements are far from trivial to satisfy. For example JML, a state-of-the-art specification language for Java, has no stable version of the run-time checker which supports all back-ends (and future ones) for Java, violating the first requirement. This is due to the fact that the JML run-time checker was designed as an extension of a proprietary Java compiler. Other tools for run-time verification such as MOP and LARVA satisfy the requirement to a certain extent. Their implementation is based on AspectJ, a compiler which extends Java with aspect-oriented programming. AspectJ can transform Java programs in bytecode form. Hence all back-ends which generate bytecode compatible with AspectJ are also supported by MOP and LARVA. This includes most, though not all, versions of the standard Sun Java compiler. However aspect-oriented programming is currently not supported by the ABS. We choose an approach

based on pre-processing. Specifications (consisting of a communication view and attribute grammar) are not added to the formal syntax of the programming language, they are put in separate files. This avoids creating multiple branches of the ABS language. In JML, specifications are added to the actual source, but in comments (so they are not part of the "logic" of the program). In MOP and LARVA, specifications are also separated from the programming language.

The input of SAGA consists of three ingredients: a communication view, an attribute grammar extended with assertions and an ABS model. The output is an ordinary ABS model which behaves the same as the input program, except that it throws an assertion failure when the current execution violates the specification. Since the resulting ABS model is an ordinary ABS model, all analysis tools [21] (including a debugging environment with visualization and a state-of-the-art cost analyzer) and back-ends which exist for the ABS can be used on it directly. The third requirement (a separation of concerns between the parser-generator and the rest of the implementation) has lead to a component-based design (Figure 8) consisting of a parser-generator component and source-code weaving component. We discuss these components, and the second requirement on performance of the generated parser, in more detail below.

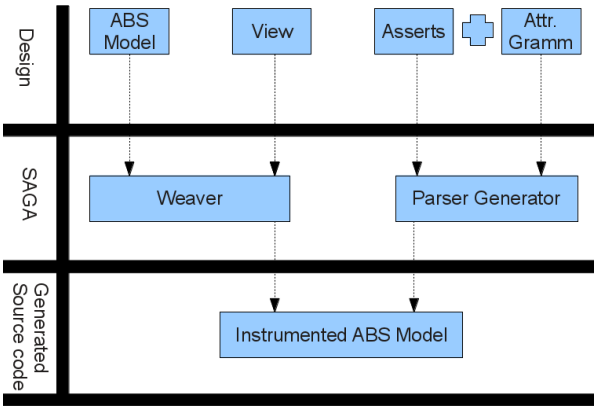

Fig. 8. SAGA tool architecture

Parser generator component The parser-generator component processes only the attribute grammar and generates a parser for it, with ABS as the target language. Parsers for attribute grammars in general take a stream of terminals as input, and output a parse tree according to the grammar productions (where non-terminal nodes are annotated with their attribute values). In our case, the attribute grammars also contains assertions, and the generated parser additionally checks that all assertions in the grammar are true.

Due to the power of general context-free grammars extended with attributes (as introduced in the seminal paper [14] by Knuth), they can be quite expen-

sive to parse. In particular, the currently best known algorithm [20] to parse context-free grammars has a time complexity of $O(n^{2.38})$ (with very huge constants), where n is the number of terminals to parse. The current best practical algorithms (with reasonably sized constants) require cubic time. Lee [15] showed that multiplication of two square Boolean matrices can be reduced to parsing context-free grammars. This gives an easy quadratic lower-bound on the time complexity of parsing (since clearly at least all elements of the two matrices must be inspected to compute the resulting product, and the two matrices have $2n^2$ entries in total). Whether this quadratic lower-bound is sharp is currently not known.

In our case, whenever a new message (asynchronous call) is added to the trace, all parse trees of all prefixes have been computed previously. The question arises how efficient the new parse trees can be computed by exploiting the parse trees of the prefixes. Unfortunately, for general context-free grammars, this cannot be done in constant time (violating the second requirement on performance). For if this was possible in constant time, parsing the full trace results in a parser which works in linear time (n terminals which all take a constant amount of time), which is lower than the theoretical quadratic lower-bound. We therefore restrict to deterministic regular attribute grammars with only inherited attributes. All grammars used in the case study have this form and parsing the new trace in such grammars can be done in constant time, since they can be translated to a finite automaton with conditions (assertions) and attribute updates as actions to execute on transitions. Parsing the new message consists of taking a single step in this automaton. Moreover for such grammars, the space complexity is also very low: it is not necessary to store the entire trace, only the attribute values of the previous trace must be stored.

Source-code weaving component The weaving component processes the communication view and the given ABS model, and outputs a new ABS model in which each call to a method appearing in the view is transformed. The transformation checks whether the method call which is about to be executed is allowed by the attribute grammar, and if this is not the case, prevents unsafe behavior by throwing an assertion failure. This transformation is invasive, in the sense that it cannot be done only locally in the body of those methods actually appearing in the view, but instead it has to be done at all call-sites (in client code). To see this, suppose that the transformation *was* done locally, say in the beginning of the method body. Due to concurrency and scheduling policies, other methods which were called at a later time could have been scheduled earlier. In such a scenario, these other methods are checked earlier than the order in which they are actually called by a client, which violates the decision (see also the previous section) to treat scheduling policies orthogonally.

The transformation is done in two steps. First, all calls to methods that occur in a communication view are isolated using pattern matching in the meta-program. We created a Rascal ABS grammar for that purpose. Second, all call-statements are preceded by code which checks that the current object is part of a named cobox (note that this check really has to be done at run-time due to the

Metrics	Java	ABS
Nr. of lines of code	6400	3300
Nr. of classes	44	40
Nr. of interfaces	2	43
Nr. of functions	N/A	80
Nr. of data types	N/A	17

Table 1. Metrics of Java and ABS of the Replication System

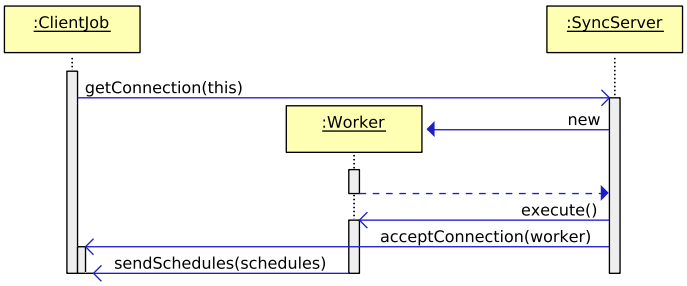

Fig. 9. Protocol violation

dynamic nature of coboxes). If this is the case, the trace is updated by taking a step in the finite automaton where additionally the assertion is checked. If there is no transition for the message from the current state, we throw an assertion error. Intuitively such an error corresponds to a protocol violation. There is one subtle point about updating the trace. If no assumptions are made about the scheduling of received messages, only updates to the trace of the calling cobox (i.e. 'send' messages in the view) can be guaranteed to be executed directly before the actual call happens. We solve this problem by using a single global trace in a dedicated cobox. This global trace is updated (using an asynchronous call) at the call site. To ensure that these updates are applied in the right order, we additionally pass in an integer which indicates the number of messages sent from the cobox, and enforce a proper scheduling using awaits based on that number.

7 Experience Report

The ABS model of the Replication System consided in the case study is a model of a part of the Fredhopper Access Server (FAS) whose current in production Java implementation has over 150,000 lines of code, of which over 6,000 lines constitute the Replication System considered here. Due to its concurrent behavior and the implementation of numerous features, the Replication System is one of the most complex parts of FAS.

Table 1 shows metrics for the actual implementation and the ABS model of the Replication System. Note that the ABS model includes model-level information such as deployment components and simulation of external inputs in the ABS model, which the Java implementation lacks. The ABS model includes

also scheduling information, as well as models of file systems and data bases, while the Java implementation leverages libraries and its API. This accounts for >1,000 lines of ABS code.

While running SAGA over the ABS model of the Replication System using the ABS Java backend, we have encountered an assertion error. The assertion error is due to the protocol violation shown in Figure 9. The sequence of messages depicted by the UML sequence diagram violates the grammar Scheduler.g shown in Figure 7. Specifically, the cobox for the `Worker` object sends the method call `SyncServer.requestListSchedules` before receiving the method call `Worker.command`. The following shows part of the implementation of `WorkerImpl` that is responsible for this violation.

```
class WorkerImpl(ClientJob job, SyncServer server, Coordinator coord)
implements Worker {
  Maybe<Command> cmd = Just(ListSchedule);
  Unit execute() {
    if (cmd == Just(ListSchedule)) {
      server!requestListSchedules(this);
    } else {
      server!requestSchedule(this, name(cmd))); }}
  Unit command(Command c) { this.cmd == Just(cmd); }}
```

The reason for the violation is that when the cobox receives the method call `Worker.execute` the above implementation does not wait receiving the method call `Worker.command` before sending the method call `SyncServer.requestListSchedules`. The reason this is possible is because the instance field `cmd` is initialized incorrectly with the value `Just(ListSchedule)` that would allow the conditional statement inside the method `acceptCoordinator` to invoke the method `SyncServer.requestListSchedules`. The following shows the correct version of this part of the implementation.

```
class WorkerImpl(ClientJob job, SyncServer server, Coordinator coord)
implements Worker {
  Maybe<Command> cmd = Nothing;
  Unit execute() {
    this.coord = coord;
    await cmd != Nothing;
    if (cmd == Just(ListSchedule)) {
      server!requestListSchedules(this);
    } else {
      server!requestSchedule(this, name(cmd))); }}
  Unit command(Command c) { this.cmd == Just(cmd); }}
```

In the correct implementation, the field `cmd` is initialized with the value `Nothing` and an **await** statement is used to ensure `cmd` is set by receiving the method call `Worker.command()` before proceeding further.

8 Conclusion

We showed that for pure asynchronous systems of coboxes which only communicate via asynchronous method calls, simple sequences of input/output messages which only refer to the targeted objects suffice for a compositional semantics. We further showed using an industrial case study how both protocol-oriented properties and data-oriented properties of such sequences can be specified conveniently in a single formalism of attribute grammars extended with assertions. Finally we developed and discussed the corresponding tool support provided by SAGA. SAGA can be obtained from `http://www.cwi.nl/~cdegouw`.

Related Work In [10] a survey is presented of behavioral interface specification languages and their use in static analysis of correctness of object-oriented programs. In particular, there exists an extensive literature on the static analysis of systems of concurrent objects. For example, in [9] a proof system for partial correctness reasoning about concurrent objects is established based on traces and class invariants. We present the first specification language for the analysis of concurrent *groups* of objects (coboxes), and implemented an efficient run-time checker. There exist many interesting approaches to run-time verification, e.g., monitoring message sequences, but all of these work in the context of Java and its low-level concurrency model based on multithreading.

For example, Martin et al. [16] introduce the Program Query Language (PQL) for detecting errors in sequences of communication events. PQL was updated last in 2006 and does not support user-defined properties of data. Allan et al. [2] develop an extension of AspectJ with a trace-based language feature called Tracematches that enables the programmer to trigger the execution of extra code by specifying a regular pattern of events in a computation trace. The underlying pattern matching involves a binding of values to free variables. Nobakht et al. [17] monitors calls and returns with the same Java Debugger Architecture that we have also evaluated in the implementation section. Their specification language is equivalent in expressive power to regular expressions. Because the grammar for the specifications is fixed, the user can not specify a convenient structure themselves, and data is not considered. Chen et al. [3] present JavaMOP, a run-time monitoring tool based on aspect-oriented programming which uses context-free grammars to describe properties of the control flow of traces. However properties on the data-flow are *predefined* built-in functions (basically AspectJ functions such as a 'target' to bind the callee and 'this' to bind the caller). LARVA is developed by Colombo et al. [5]. The specification language has an imperative flavour: users define a finite state machine to define the allowed traces (i.e. one has to 'implement' a regular expression themselves). Data properties are supported in a limited manner, by enriching the state machine with conditions on method parameters or return values (not on sequences of them).

DeLine and Fähndrich [8] propose a statically checkable typestate system for object-oriented programs. Typestate specifications of protocols correspond to finite state machines, data and assertions are not considered in their approach.

Future Work For practical reasons, good error reporting is essential. Note however that since error reporting, for example in case of assertion failures, prints to the screen (and consequently relies on low-level I/O details), it is not back-end independent. Using the ABS foreign language interface, it is possible to execute native Java or Maude code which implements the error reporting. As a relatively simple first step, we could for instance use SDEdit, a sequence diagram editor already used in the ABS, to visualize traces violating the grammars. Since traces tend to be large, finding relevant abstractions of the trace is crucial here.

Currently SAGA supports determinstic regular grammars with just inherited attributes. Such grammars can be incrementally parsed. This immediately suggest another future line of work: is there a larger class of grammars which can be parsed incrementally?

As the final direction of future work we would like to investigate ways to control the complexity of extensions of the modeling language including futures and promises (in the Cobox concurrency model).

References

1. G. Agha. Actors: A model of concurrent computation in distributed systems. *MIT Press, Cambridge, MA, USA*, 1990.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA*, pages 345–364, 2005.
3. F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.
4. L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.
5. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *SEFM*, pages 33–37, 2009.
6. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.
7. F. S. de Boer, S. de Gouw, E. B. Johnsen, and P. Y. H. Wong. Run-time checking of data- and protocol-oriented properties of java programs: An industrial case study. In *SAC, to appear*, 2013.
8. R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
9. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. Log. Algebr. Program.*, 81(3):227–256, 2012.
10. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

12. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.
13. P. Klint, T. van der Storm, and J. Vinju. Rascal: a domain specific language for source code analysis and manipulation. In A. Walenstein and S. Schupp, editors, *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 168–177, 2009.
14. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
15. L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.
16. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPLSLA*, 2005.
17. B. Nobakht, M. M. Bonsangue, F. S. de Boer, and S. de Gouw. Monitoring method call sequences using annotations. In *FACS*, pages 53–70, 2010.
18. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.
19. M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.
20. L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.
21. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The abs tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.