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 verification is one of the most useful techniques for detecting faults. 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 efficient 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 [15] 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 modeling language described in [11]. 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.

ABS (Abstract Behavioral Specification language) is a novel language based on coboxes for modeling and analysis of complex distributed systems. It is a fully executable language with code generators for Java, Maude and Scala. In [10] a formal semantics of ABS was introduced based on asynchronous messages between coboxes. However, as of yet, no formal method for specifying and run-time verifying traces of such asynchronous messages has been developed. The main contribution of this paper is tool support for the efficient run-time verification

R.M. Hierons, M.G. Merayo, and M. Bravetti (Eds.): SEFM 2013, LNCS 8137, pp. 259–273, 2013.
© Springer-Verlag Berlin Heidelberg 2013

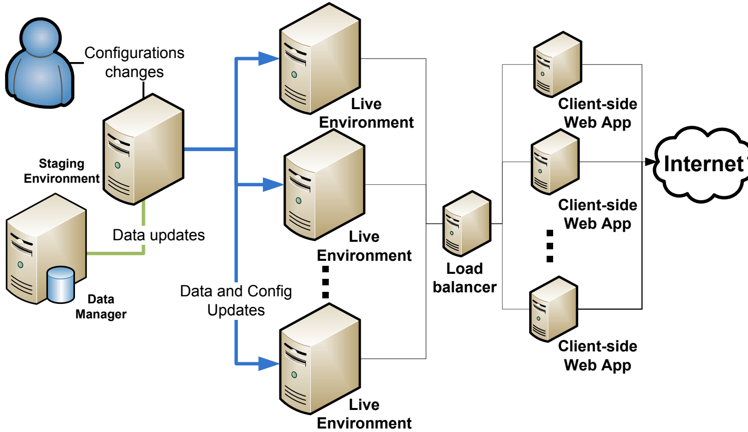

Fig. 1. An example FAS deployment

of asynchronous message passing between coboxes, independent from any backend. This latter requirement is important because in general the analysis of a particular backend is complicated by low-level implementation details. Further, it allows to generalize the analysis to all (including future) backends.

Run-time verification 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 show how to use attribute grammars extended with assertions to specify and verify (at run-time) properties of the messages sent between coboxes. To this end, we first improve the efficiency of the run-time verification tool SAGA [6], which smoothly integrates both data- and protocol-oriented properties of message sequences. Both time and space complexity of SAGA is linear in the size of the message sequence. Further we extend it to support design-by-contract for coboxes. We illustrate the effectiveness of our method by an industrial case study from the eCommerce software company Fredhopper.

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

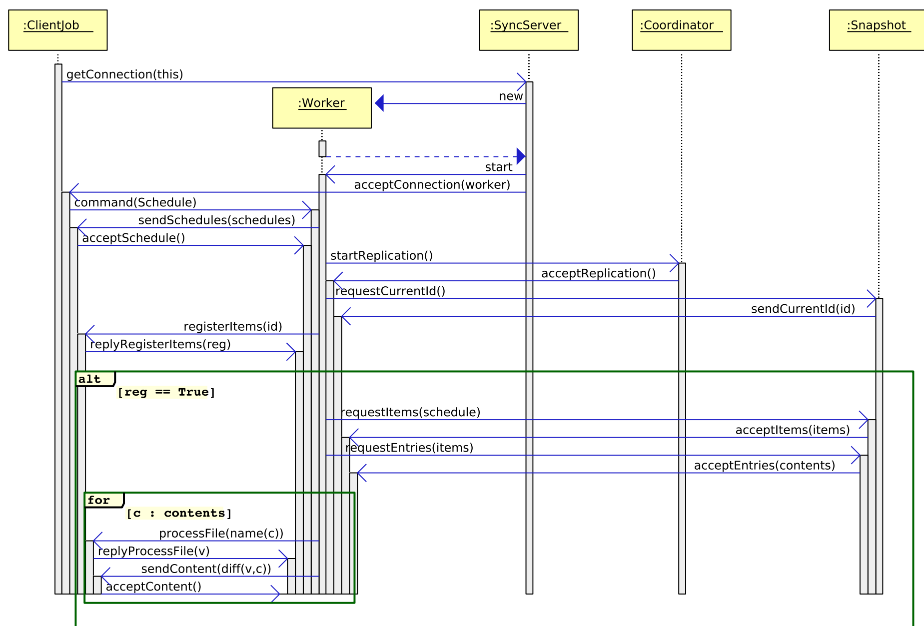

Fig. 2. Replication interaction

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.

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 with 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 (`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

We formally describe coboxes by means of a modeling language which is based on the *Abstract Behavioral Specification* language [10]. Throughout the paper we refer to our own modeling language by ACOG (pure Actor-based Concurrent Object Groups). ACOG 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. ACOG generalizes the concurrency model of Creol [11] from single concurrent objects to concurrent object groups (coboxes). As in [15] coboxes encapsulate synchronous, multi-threaded, shared state computation on a single processor. In contrast to thread-based concurrency, 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 writing concurrent programs in a much less error-prone way than in a thread-based model and makes ACOG models suitable for static analysis. In our dialect, unlike in [15], coboxes communicate only via pure asynchronous messages, and as such form an actor-based model as initially introduced by [1] and further developed in [16].

The following fragment of `ClientJobImpl` illustrates 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 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 to arrive. The next set of schedules is set by invoking the method `sendSchedules` on the ClientJob.

4 Behavioral Interfaces for Coboxes

In this section we introduce *attribute grammars* extended with assertions to specify and verify properties of the traces generated between coboxes. As such, extended attribute grammars provide a new formalism for contracts in general, and coboxes in particular. In contrast to classes or interfaces, coboxes are run-time entities which do not have a single fixed interface[1]. 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.

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

4.1 Communication Views

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

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

The semantics of the language remain unchanged. Note that the same cobox name can be shared among several coboxes (i.e. is in general not unique) since different cobox creation statements can specify the same cobox 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 3 shows an example communication view associated with all coboxes named WorkerGroup. Formally a communication view is a

```
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. 3. Communication View

partial 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 event 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.

Note that in this asynchronous setting we can distinguish three different events: sending a message (at the call-site), receiving the message in the queue (at the callee-site), and scheduling the message for execution (i.e. the point in time when the corresponding method starts executing). By the asynchronous nature of the ABS, we cannot detect in the ABS itself when a message has been put into the queue. Therefore we restrict to the other two events. Since we implement the run-time checker independently from any back-end (see also Section 5), we are forced to use the ABS itself for the detection of the observable events.

The `send` keyword identifies calls from objects in the WorkerGroup to methods of objects in another cobox (in other words: methods required by an object in the WorkerGroup). Conversely, the keyword `receive` identifies the scheduling of calls from another cobox to an object in a WorkerGroup. 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`.

4.2 Grammars

In this subsection we describe how properties of the set of allowed traces of a cobox can be 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 retrieve 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 3. The productions of the grammar underlying the attribute grammar in Figure 4 specify the legal orderings of these messages named in the view. For example, the productions

$$\begin{aligned} S &::= \epsilon \mid st\ \ T \\ T &::= \epsilon \mid id\ \ U \end{aligned}$$

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 consisting 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.[2] For example, in the following production, the attribute 'w' for the non-terminal 'T' is defined.

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

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

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

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

expresses that the 'id' passed as a parameter to the method 'registerItems' (represented in the grammar by the terminal pr.`id;`) must be the same as the 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 4 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.

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

Fig. 4. Attribute Grammars

To further illustrate the above concepts, we consider an additional behavioral interface for the WorkerGroup cobox. To allow users to make changes to 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 scheduling. Here is an informal description of the property, where Figure 5 presents the

```
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. 5. Communication View for Scheduling

```
S ::= ϵ | cm  T (T.c = cm.c;)
T ::= ϵ | gt  {assert T.c != ListSchedule &&
                 gt.n == name(T.c);} U (U.c = T.c;)
        | lt  {assert T.c == ListSchedule;} U (U.c = T.c;)
U ::= ϵ | sn  {assert sn.ss != EmptySet;} V (V.c = U.c;)
V ::= ϵ | st  {assert V.c != ListSchedule;}
```

Fig. 6. Attribute Grammar for Scheduling

communication view capturing the relevant messages and Figure 6 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.

5 Implementation

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 [18] (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. 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. This has lead to a component-based design (Figure 7) 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.

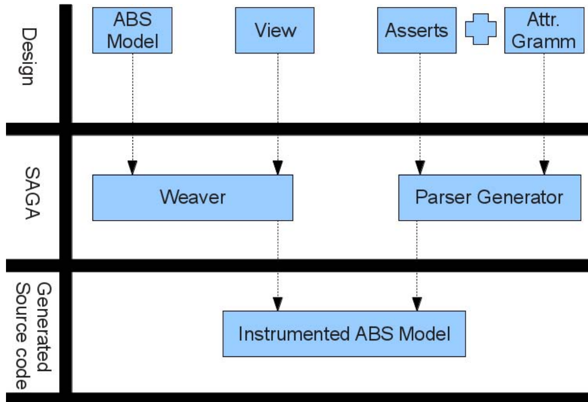

Fig. 7. SAGA tool architecture

Component for parsing deterministic attribute grammars with inherited attributes. This 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 contain assertions, and the generated parser additionally checks that all assertions in the grammar are true.

Due to the power of general context-free grammars (even without attributes), they can be quite expensive to parse. By combining results of [17] and Lee [12] we can deduce the time complexity of parsing n tokens lays between $O(n^2)$ and $O(n^{2.38})$. However, 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. 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 our attention 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 inserts code which 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. In contrast to receive events, the transformation for send events 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.

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 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.

6 Experience Report

In order to understand the Fredhopper Access Server (FAS), a system with over 150,000 lines of Java code, suitable abstractions are crucial. We developed an ACOG model which describes the behavior of the replication system, a crucial (and one of the most complex) component in the FAS. We then specified and checked this behavior by means of attribute grammars with SAGA.

Table 1 shows metrics for the Java implementation and the ACOG model of the Replication System (without the attribute grammar). The figures in the table illustrate the expressive power of the modeling language: the ACOG model is half the size of the Java implementation. Additionally the ACOG model includes model-level information such as deployment components and simulation

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

Metrics	Java	ACOG
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

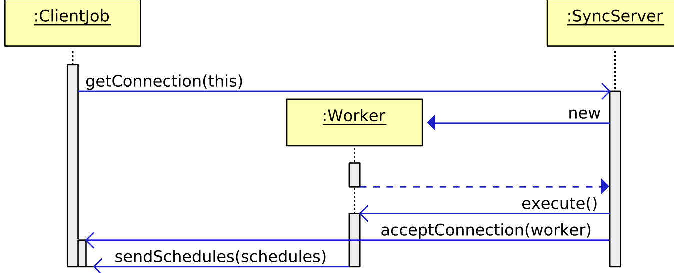

Fig. 8. Protocol violation

of external inputs in the ACOG model, which the Java implementation lacks. The ACOG 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 ACOG code.

We detected a crucial protocol violation while running SAGA over the ACOG model of the Replication System (see Figure 8). The sequence of messages depicted by the UML sequence diagram violates the grammar Scheduler.g shown in Figure 6. 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 execute 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.

7 Conclusion

We showed using an industrial case study how both protocol-oriented properties and data-oriented properties of message sequences sent between coboxes can be specified conveniently in a single formalism of attribute grammars extended with assertions. Moreover we developed and discussed the corresponding tool support provided by SAGA. SAGA can be obtained from http://www.cwi.nl/~cdegouw.

Related Work. In [9] 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 [8] 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. This paper builds on the previous work [6], which integrates (in a single formalism) both data- and protocol-oriented properties of message sequences of single-threaded Java programs. Here we extend this work to a concurrent modeling language, which requires a very different tool architecture, and add support for incremental parsing of message sequences with a linear space- and time-complexity. There exist many interesting approaches to run-time verification, e.g., monitoring message sequences, but all of these approaches only work in the context of Java and its low-level concurrency model based on multithreading.

For example, Martin et al. [13] 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. [14] monitors calls and returns using the Java Debugger Architecture. 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 JavaMOP does not integrate data-oriented properties for use in design-by-contract a la JML. General data-oriented properties can only be specified by the injection of Java **assert**-statements using AspectJ, essentially bypassing JavaMOP. Moreover even this manual injection can only be used to specify a data-property of the single last message sent/received, not for data properties of the full history. As such, JavaMOP provides no direct and high-level support for data-oriented properties. 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 manually 'implement' a parser for the regular expression). 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 [7] 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 deterministic 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. Agha, G.: Actors: A model of concurrent computation in distributed systems. MIT Press, Cambridge (1990)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. SIGPLAN Not. 40(10), 345–364 (2005)
3. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. SIGPLAN Not. 42(10), 569–588 (2007)
4. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes 31(3), 25–37 (2006)
5. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: SEFM 2005, pp. 33–37 (2009)
6. de Boer, F.S., de Gouw, S., Johnsen, E.B., Wong, P.Y.H.: Run-time checking of data- and protocol-oriented properties of java programs: An industrial case study. In: SAC (to appear, 2013)
7. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
8. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. J. Log. Algebr. Program. 81(3), 227–256 (2012)
9. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. 44(3), 16:1–16:58 (2012)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
11. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. SSM 6(1), 35–58 (2007)
12. Lee, L.: Fast context-free grammar parsing requires fast boolean matrix multiplication. J. ACM 49(1), 1–15 (2002)
13. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using pql: a program query language. SIGPLAN Not. 40(10), 365–383 (2005)
14. Nobakht, B., Bonsangue, M.M., de Boer, F.S., de Gouw, S.: Monitoring method call sequences using annotations. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 53–70. Springer, Heidelberg (2012)
15. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
16. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. 63(4), 385–410 (2004)
17. Valiant, L.G.: General context-free recognition in less than cubic time. J. Comput. Syst. Sci. 10(2), 308–315 (1975)
18. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT 14(5), 567–588 (2012)