# Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study

Frank S. de Boer[1,2], Stijn de Gouw[1,2],
Einar Broch Johnsen[3], Andreas Kohn[4], and Peter Y. H. Wong[4]

[1] CWI, Amsterdam, The Netherlands
[2] Leiden University, The Netherlands
[3] University of Oslo, Norway
[4] Fredhopper B.V., Amsterdam, The Netherlands

**Abstract.** Run-time assertion checking is one of the useful techniques for detecting faults, and can be applied during any program execution context, including debugging, testing, and production. In general, however, it is limited to checking state-based properties. We introduce SAGA, a general framework that provides a smooth integration of the specification and the run-time checking of both data- and protocol-oriented properties of Java classes *and interfaces*. We evaluate SAGA, which combines several state of the art tools, by conducting an industrial case study from an eCommerce software company Fredhopper.

## 1 Introduction

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 [7]. Compared to program logics, run-time assertion checking emphasizes *executable specifications*. Assertions in general are clearly not executable in the sense that one cannot decide whether they hold (in the presence of unbounded quantification). As a result, for run-time assertion checking one has to restrict the class of assertions to executable ones. Whereas program logics are generally applied statically to cover all possible execution paths, which is in general undecidable, run-time assertion checking is a fully automated, on-demand validation process which applies to the actual runs of the program.

By their very nature, assertions are state-based in that they describe properties of the program variables, e.g. fields of classes and local variables of methods. In general, assertions as supported for example by the Java programming language or the Java Modeling Language (JML) [3] cannot be used to specify the *interaction protocol* between objects, in contrast to other formalisms such as message sequence charts and UML sequence diagrams. Consequently, existing state-of-the-art program logics for Java are not suited for proving protocol prop-

erties. Moreover state-based assertions cannot be used to specify interfaces since interfaces do not have a state[5].

The main contribution of this paper is twofold. Firstly, we introduce SAGA (Software trace Analysis using Grammars and Attributes), 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 Java classes *and interfaces.* SAGA combines four different components: a state-based assertion checker, a monitoring tool, a meta-programming tool and a parser generator. Aspect-oriented programming is tailored for monitoring, and in contrast to transformations source of Java code or debugger-based solutions [9] it is designed for high performance applications and supports the monitoring of precompiled libraries for which no source code is available. The tool can be used for run-time checking of any Java program, which requires specific support for the main features listed in Table 1, as discussed in more detail in the following section. Secondly, we evaluate SAGA by conducting an industrial case study from the eCommerce software company Fredhopper.

| Constructors |
|---|
| Inheritance |
| Dynamic Binding |
| Overloading |
| Static Methods |
| Access Modifiers |

**Table 1.** Supported features

The basic idea underlying SAGA is the representation of message sequences as words of a language generated by a grammar. Grammars allow, in a declarative and highly convenient manner, the description of the *protocol structure* of the communication events. However, the question is how to integrate such grammars with the run-time checking of assertions, and how to describe the *data flow* of a message sequence, i.e., the properties of the data communicated. We propose a formal modeling language for the specification of sequences of messages in terms of *attribute grammars* [14]. Attribute grammars allow the high-level specification of the data-flow of message sequences (e.g., their length) in terms of user-defined attributes of non-terminals. SAGA supports the run-time checking of assertions about these attributes (e.g., that the length of a sequence is bounded). This involves parsing the generated sequences of messages. These sequences themselves are recorded by means of a fully automated instrumentation of the given program by AspectJ[6].

[5] JML uses model variables for interface specifications. However, a separate represents clause is needed for a full specification, and such clauses can only be defined once an implementation has been given (and is not implementation independent).

[6] `www.eclipse.org/aspectj`

## 2 The Modeling Framework

Abstracting from implementation details (such as field values of objects), an execution of a Java program can be represented by its *global communication history*: the sequence of *messages* corresponding to the invocation and completion of (possibly static) methods. Similarly, the execution of a single object can be represented by its *local communication history*, which consists of all messages sent and received by that object. The behavior of a program (or object) can then be defined as the set of its allowed histories. Whether a history is allowed depends in general both on data (the contents of the messages, e.g. parameter and return values of method calls) and protocol (the order between messages). The question arises how such allowed sets of histories can be defined conveniently. In this section we show how attribute grammars provide a powerful and declarative way to define such sets. We use the interface of the Java `BufferedReader` (Figure 1) as a running example to explain the basic modeling concepts. In particular, we formalize the following property:

> The `BufferedReader` may only be closed by the same object which created it, and reads may only occur between the creation and closing of the `BufferedReader`.

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

**Fig. 1.** Methods of the BufferedReader Interface

As a naive first step one might be tempted to define the behavior of `BufferedReader` objects simply in terms of 'call-m($\overline{T}$)' and 'return-m($\overline{T}$)' messages of all methods 'm' in its interface, where the parameter types $\overline{T}$ are included to distinguish between overloaded methods (such as `read`). However, interfaces in Java contain only signatures of provided methods: methods where the `BufferedReader` is the callee. Calls to these methods correspond to messages received by the object. In general the behavior of objects also depends on messages sent by that object (i.e. where the object is the caller), and on the particular constructor (with parameter values) that created the object. Moreover it

is often useful to select a particular subset of method calls or returns, instead of using calls and returns to all methods (a partial or incomplete specification). Finally in referring to messages it is cumbersome to explicitly list the parameter types. A *communication view* addresses these issues.

### 2.1 Communication View

A communication view is a partial mapping which associates a name to each message. Partiality makes it possible to filter irrelevant events and message names are convenient in referring to messages.

Suppose we wish to specify that the `BufferedReader` may only be closed by the same object which created it, and that reads may only occur between the creation and closing of the `BufferedReader`. This is a property which must hold for the local history of all instances of `java.util.BufferedReader`. The communication view in Figure 2 selects the relevant messages and associates them with intuitive names: *open*, *read* and *close*.

```
local view BReaderView grammar BReader.g
specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

**Fig. 2.** Communication view of a BufferedReader

All return messages and call messages methods not listed in the view are filtered. Note how the view identifies two different messages (calls to the overloaded read methods) by giving them the same name *read*. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g. methods of other interfaces or classes) are also supported. Since such messages are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example consider the following message:

```
call void C.m() out
```

If we would additionally include the above message in the communication view, all call-messages to the method `m` of class `C` sent by a `BufferedReader` would be selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods

and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Besides normal methods, communication views can contain signatures of constructors (i.e. the messages named *open* in our example view). Incoming calls to provided constructors raise an interesting question: what would happen if we select such a message in a local history? At the time of the call, the object has not even been created yet, so it is unclear which `BufferedReader` object receives the message. We therefore only allow return-messages of provided constructors (clearly required constructors do not pose the same problem, and consequently we allow selecting both calls and returns to required constructors), and for convenience omit **return**. Alternatively one could treat constructors like static methods, disallowing incoming call-messages to constructors in local histories altogether. However this makes it impossible to express certain properties (including the desired property of the `BufferedReader`) and has no advantages over the approach we take.

Java programs can distinguish methods of the same name only if their parameter types are different. Communication views are more fine-grained: methods can be distinguished also based on their return type or their access modifiers (such as **public**). For instance, consider a scenario with suggestively named classes `Base` and three subclasses `Sub1`, `Sub2` and `Sub3`, all of which provide a method `m`. The return type of `m` in the `Base`, `Sub1` and `Sub2` classes is the class itself (i.e. `Sub1` for `m` provided by `Sub1`). In the `Sub3` class the return type is `Sub1`. To monitor calls to `m` only with return type `Sub1`, simply include the following event in the view:

```
call Sub1 C.m() messagename
```

Local communication views, such as the one above, selects messages sent and received by *a single object* of a particular class, indicated by 'specifies java.util.BufferedReader'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views. Figure 3 shows a global view which selects all returns of the method `m` of the `Ping` class or interface or any of its subclasses, and all calls of the `Pong` class (or interface) or its subclasses. Note that communication views do not distinguish instances of the same class (e.g. calls to 'Ping' on two different objects of class 'Ping' both get mapped to the same terminal 'ping'). Different instances can be distinguished in the grammar using the built-in attributes 'caller' or 'callee'.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. See table 1 for a list of supported features.

```
global view PingPong grammar pingpong.g {
  return void Ping.m() ping,
  call void Pong.m() pong
}
```

**Fig. 3.** Global communication view

### 2.2 Grammars

Context-free grammars provide a convenient way to define the protocol behavior of the allowed histories. The context-free grammar underlying the attribute grammar in Figure 4 generates the valid histories for `BufferedReader`, describing the prefix closure of sequences of the terminals 'open', 'read' and 'close' as given by the regular expression (open read* close). In general, the message names form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of messages. In our approach, a communication history is valid if and only if it and all its prefixes are generated by the grammar.

For a justification of this approach, see the next discussion section. While context-free grammars provide a convenient way to specify the *protocol structure* of the valid histories, 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 the valid histories. To that end, we extend the grammar with attributes. Each terminal symbol has *built-in* attributes named **caller**, **callee** and the parameter names for respectively the object identities of the caller, callee and actual parameters. Terminals corresponding to method returns additionally have an attribute **result** containing to the return value. In summary, the (built-in) attributes of terminals are determined from the method signatures. Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. However the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the history. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Figure 4 a user-defined synthesized attribute 'c' for the non-terminal 'C' is defined to store the identity of the object which closed the `BufferedReader` (and is **null** if the reader was not closed yet). Synthesized attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the 'c' attribute is not set in the productions of the start symbol 'S'.

The assertion allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the paper the start symbol in any grammar is named 'S'. For clarity, attribute definitions are written between parentheses '(' and ')' whereas assertions over these attributes are surrounded by braces '{' and '}'.

$$
\begin{array}{lll}
S ::= \mathit{open}\ C_1 & \{\textsf{\textbf{assert}}\ \texttt{(open.}\textsf{\textbf{caller}}\ \texttt{==}\ \textsf{\textbf{null}}\ \texttt{||}\ \texttt{open.}\textsf{\textbf{caller}}\ \texttt{==}\ C_1\texttt{.c}\ \texttt{||} \\
 & \quad C_1\texttt{.c == }\textsf{\textbf{null}}\texttt{);}\} \\
\quad \mid\ \epsilon & \\
C ::= \mathit{read}\ C_1 & (C\texttt{.c = }C_1\texttt{.c;}) \\
\quad \mid\ \mathit{close}\ S & (C\texttt{.c = close.}\textsf{\textbf{caller}}\texttt{;}) \\
\quad \mid\ \epsilon & (C\texttt{.c = }\textsf{\textbf{null}}\texttt{;})
\end{array}
$$

**Fig. 4.** Attribute Grammar which specifies that 'read' may only be called in between 'open' and 'close', and the reader may only be closed by the object which opened it.

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal 'call-m' can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears.

Attribute grammars in combination with assertions cannot express protocol that depend on data. Such protocols are common, for instance, the method `next` of an `Iterator` may not be called if directly `hasNext` was called directly before and *returns* **false**. To express protocols depending on data we consider attribute grammars enriched by *conditional productions* [18]. In such grammars, a production is chosen only when the given condition (a **boolean** expression over the inherited attributes) for that production is true. Hence conditions are evaluated before any of the symbols in the production are parsed, before synthesized attributes of the non-terminals appearing in the production are set and before assertions are evaluated. In contrast to assertions, conditions in productions affect the parsing process. The Worker.g grammar in the case study contains a conditional production for the 'T' non-terminal.

### 2.3 Discussion

We now briefly motivate our choice of attribute grammars extended by assertions as specifications and discuss its advantages over alternative formalisms.

Instead of context-free grammars, we could have selected push-down automata to specify protocol properties (formally these have the same expressive power). Unfortunately push-down automata cannot handle attributes. An extension of push-down automata with attributes results in a register machine. From a user perspective, the declarative nature and higher abstraction level of grammars (compared to the imperative and low-level nature of automata) makes them much more suitable than automata as a *specification* language. In fact, a push-down automaton which recognizes the same language as a given grammar is an *implementation* of a parser for that grammar.

Both the BufferedReader above and the case study use only regular grammars. Since regular grammars simplify parsing compared to context-free grammars, the question arises if we can reasonably restrict to regular grammars.

Unfortunately this rules out many real-life use cases. For instance, the following grammar in EBNF specifies the valid protocol behavior of a stack:

$$S ::= (push\ S\ pop\ ?)^*$$

It is well-known that the language generated by the above grammar is not regular, so regular grammars (without attributes) cannot be used to enforce the safe use of a stack. It is possible to specify the stack using an attribute which counts the number of pushes and pops:

$$\begin{array}{lll} S ::= S_1\ push & (S.cnt = S_1.cnt + 1) & \\ \quad\mid\ \ S_1\ pop & (S.cnt = S_1.cnt - 1)\{assert\ S.cnt >= 0;\} \\ \quad\mid\ \ \epsilon & (S.cnt = 0) & \end{array}$$

The resulting grammar is clearly less elegant and less readable: essentially it encodes (instead of directly expresses, as in the grammar above) a protocol-oriented property as a data-oriented one. The same problem arises when using regular grammars to specify programs with recursive methods. Thus, although theoretically possible, we do not restrict to regular grammars for practical purposes.

Ultimately the goal of run-time checking safety properties is to prevent unsafe ongoing behavior. To do so, errors must be detected as soon as they occur and the monitor must *immediately* terminate the system: it cannot wait until the program ends to detect errors. In other words, the monitor must decide *after every event* whether the current history is still valid. The simplest notion of a valid history (one which should not generate any error) is that of a word generated by the grammar. One way of fulfilling the above requirement, assuming this notion of validity, is to restrict to prefix-closed grammars. Unfortunately it's not possible to decide whether a context-free grammar is prefix-closed. The following lemmas formalize this result:

**Lemma 1.** *Let $L_M$ be the set of all accepting computation histories*[7] *of a Turing Machine M. Then the complement $\overline{L_M}$ is a context-free language.*

*Proof.* See [20].

**Lemma 2.** *It is undecidable whether a context-free language is prefix-closed.*

*Proof.* We show how the halting problem for $M$ (which is undecidable) can be reduced to deciding prefix-closure of $\overline{L_M}$. To that end, we distinguish two cases:

1. $M$ does not halt. Then $L_M$ is empty so $\overline{L_M}$ is universal and hence prefix-closed.

[7] A computation history of a Turing Machine is a sequence $C_0\#C_1\#C_2\#\ldots$ of configurations $C_i$. Each configuration is a triple consisting of the current tape contents, state and position of the read/write head. Due to a technicality, the configurations with an odd index must actually be encoded in reverse.

2. $M$ halts. Then there is an accepting history $h \in L_M$ (and $h \notin \overline{L_M}$). Extend $h$ with an illegal move (one not permitted by $M$) to the configuration $C$, resulting in the history $h\#C$. Clearly $h\#C$ is not a valid accepting history, so $h\#C \in \overline{L_M}$. But since $h \notin \overline{L_M}$, $\overline{L_M}$ is not prefix-closed.

Summarizing, $M$ halts if and only if $\overline{L_M}$ is not prefix-closed. Thus if we could decide prefix-closure of the context-free language (lemma 1) $\overline{L_M}$, we could decide whether $M$ halts.

Since prefix-closure is not a decidable property of grammars (not even if they don't contain attributes) we propose the following alternative definition for the valid histories. A communication history is valid if and only if it and all its prefixes are generated by the grammar. Note that this new definition naturally fulfills the above requirement of detecing errors after every event. And furthermore this notion of validity is decidable assuming the assertions used in the grammar are decidable. As an example of this new notion of validity, consider the following modification of the above grammar:

$$
\begin{array}{rlll}
T & ::= & S & \{assert\ S.cnt >= 0;\} \\
S & ::= & S_1\ push & (S.cnt = S_1.cnt + 1) \\
 & | & S_1\ pop & (S.cnt = S_1.cnt - 1) \\
 & | & \epsilon & (S.cnt = 0)
\end{array}
$$

Note that the history *push pop* is a word generated by this grammar, but not its prefix *pop*, which as such will generate an error (as required). Note that thus in general invalid histories are guaranteed to generate errors. On the other hand, if a history generates an error all its extensions are therefore also invalid.

Observe that our approach monitors only safety properties ('prevent bad behavior'), not liveness ('something good eventually happens'). This restriction is not specific to our approach: liveness properties in general cannot be rejected on any finite prefix of an execution, and monitoring only checks finite prefixes for violations of the specification. Most liveness properties fall in the class of the non-monitorable properties [2, 19]. However it *is* possible to ensure liveness properties for terminating programs: they can then be reformulated as safety properties. For instance, suppose we want to guarantee that a method `void m()` is called before the program ends. Introduce the following global view

```
global view livenessM {
  call void C.m() m,
  return static void C.main(String[]) main
}
```

The occurence of the 'main' event (i.e. a return of the main method of the program) signifies the program is about to terminate. Define the EBNF grammar
$S ::= \epsilon \mid m \mid m+\ main$
(where '+' stands for one or more repetitions). This grammar achieves the desired effect since the only terminating executions allowed are those containing `m`. In

local views a similar effect is obtained by including the method `finalize` instead of `main`.

# 3 Tool Architecture

In this section we describe the tool architecture of the run-time assertion checker. The checker integrates four different components: a state-based assertion checker, a parser generator, a monitoring tool and a general tool for meta-programming. These components are traditionally used for very diverse purposes and normally don't need to interact with each other. We investigate requirements needed to achieve a seamless integration, motivated by describing the workflow of the run-time checker. Finally we instantiate the components with actual tools and evaluate them.

## 3.1 Workflow

A user starts executing a Java class with a main statement. Suppose that during execution, a method listed in a communication view is called. The history should be updated to reflect the addition of the method call. Thus the question arises how to represent the history. A **meta-program** generates for each message in the communication view a class (subsequently called 'token classes') containing the following fields: the object identitities of the *caller* and *callee*, the actual parameter values and for return messages additionally a field `result` to store the return value. The history can then be represented as a Java `List` of instances of token classes.

Next the **monitoring tool** should update the history whenever a call or return listed in a view occurs. Thus the monitoring tool should be capable of executing user-defined code directly before method calls and directly after method returns. Moreover it must be able to read the identity of the callee, caller and parameters/return-value.

After the history is updated the SAGA must decide whether it still satisfies the specification (the attribute grammar). Observe that a communication history can be seen as a sequence of tokens (in our setting: communication events). Since the attribute grammar together with the assertions generate the language of all valid histories, checking whether a history satisfies the specification reduces to deciding whether the history can be parsed by a parser for the attribute grammar, where moreover during parsing the assertions must evaluate to true.

Therefore the **parser generator** creates a parser for the given attribute grammar. Since the history is a list of token class objects, the parser must support parsing streams of user-defined token types. As the (user-defined) attributes of non-terminals in the grammar are defined in terms of built-in attributes of terminals (recall those are for example, actual parameter values), and clearly the built-in attributes are Java objects, the user-defined attributes must also be Java objects. Consequently the target language for the parser generator must be Java, and it must support executing user-defined Java code to define the

attribute value in rule actions. The use of Java code to define attribute values ensures they are computable. Furthermore, assertions are allowed in-between any two (non)-terminals, thus the parser generator should support user-defined actions between arbitrary grammar symbols. Once the parser is generated, it is triggered whenever the history of an object is updated.

During parsing, the **state-based assertion checker** proceeds to evaluate the assertions in the grammar on the newly computed attribute values. The result is either a parse or assertion error, which indicates that the current communication history has violated the specification in the attribute grammar, or a parse tree with new attribute values.

### 3.2 Implementation

In this section we instantiate each of the four different components (meta-programming, monitoring tool, parser generator and state-based run-time assertion checker) with a state of the art tool. We report on our experiences with the particular tools and discuss the extend to which the previously formulated requirements are fulfilled.

Rascal [13] is a powerful tool-supported meta-programming language tailored for program analysis, program transformation and code generation. We have written a Rascal program of approximately 600 lines in total which generates the token class for each message in the view, and generates glue code to trigger the AspectJ and parser at the appropriate times. Overall our experience with Rascal was quite positive: its powerful parsing, pattern matching and transforming concrete syntax features were indispensable in the implementation of SAGA.

As the parser generator we tested ANTLR [17], a state of the art parser generator. It generates fast recursive descent parsers for Java, has direct support for both synthesized and inherited attributes, it supports grammars in EBNF form and most importantly allows a custom stream of token classes. It even supports *conditional productions*: such productions are only taken during parsing whenever an associated Boolean expression (the condition) is true. Attribute grammars with conditional productions express protocols that depend on data, and typically are not context-free. The worst-case time complexity of any parser ANTLR generates is quadratic in the number of tokens to parse. The main drawbacks of ANTLR are that it can only handle LL(*) grammars[8], and its lack of support for incremental parsing, though support for incremental is planned by the ANTLR developers. An incremental parser computes a parse tree for the new history based on the parse trees for prefixes of the history. In our setting, since the attribute grammar specifies invariant properties of the ongoing behavior, a new parse tree is computed after each call/return, hence parse trees for all prefixes of the current history can be exploited for incremental parsing [11]. We

[8] A strict subset of the context-free grammars. Left-recursive grammars are not LL(*). A precise definition can be found in [17].

have not been able to find any Java parser generator which supported general context-free grammars and incremental parsing of attribute grammars.

We have tested two state-based assertion languages: standard Java assertions and the Java Modeling Language (JML). Both languages suffice for our purposes. JML is far more expressive than the standard Java assertions, though its tool support is not ready for industrial usage. In particular, the last stable version of the JML run-time assertion checker dates back over 8 years, when for example generics were not supported yet. The main reason is that JML's run-time assertion checker only works with a proprietary implementation of the Java compiler, and unsurprisingly it is costly to update the propietary compiler each time the standard compiler is updated. This problem is recognized by the JML developers [4]. OpenJML [9], a new pre-alpha version of the JML run-time assertion checker integrates into the standard Java compiler, and initial tests with it provided many valuable input for real industrial size applications. See the Sourceforge tracker for the kind of issues we have encountered when using OpenJML.

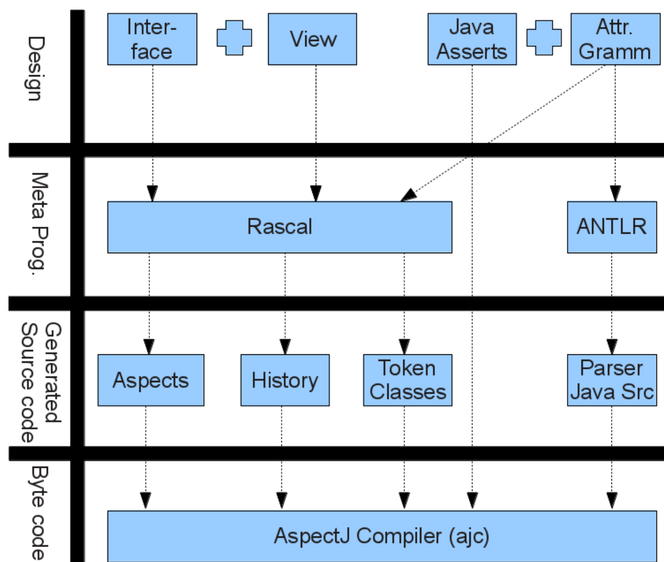



**Fig. 5.** SAGA Tool Architecture

*Aspects* AspectJ is tailored for monitoring. It can intercept method calls and returns conveniently with pointcuts, and weave in user-defined code (advices) which is executed before or after the intercepted call. In our case the pointcuts

[9] `jmlspecs.sourceforge.net`

correspond to the calls and returns of the messages listed in the communication view. The advice consists of code which updates the history. The code for the aspect is generated from the communication view automatically by the Rascal meta-program. Advice is woven into Java source code, byte code or at class load-time fully automatically by AspectJ. We use the inter-type declarations of AspectJ to store the local history of an object as a field in the object itself. This ensures that whenever the object goes out of scope, so does its history and consequently reduces memory usage. Clearly the same does not hold for global histories, which are stored inside a separate Aspect class. Figure 6 shows a generated aspect. The second and third line specify the relevant method. The fourth line binds variables ('clr', 'cle', ...) to the appropriate objects. The fifth line ensures that the aspect is applied only when Java assertions are turned on. Assertions can be turned on or off for each communication view individually. The fifth line contains the advice that updates the history. Note that since the event came was defined in a local view, the history is treated as a field of the callee (and will not persist in the program indefinitely but rather is be garbage collected as soon as callee object itself is).

```
    /* call int read(char[] cbuf, int off, int len); */
before(Object clr, BufferedReader cle, char[] cbuf, int off, in len):
 (call( int *.read(char[], int, int))
  && this(clr) && target(cle) && args(cbuf, off, len)
  && if(BReaderHistoryAspect.class.desiredAssertionStatus() )) {
    cle.h.update(new call_push(clr, cle, cbuf, off, len));
}
```

**Fig. 6.** Aspect for the event 'call int read(char[] cbuf, int off, int len)'

We have investigated two alternatives for the monitoring component not based on aspect-oriented programming: Rascal and Sun's implementation of the Java Debugging Interface. With Rascal one can weave advice by defining a transformation on the actual Java source code of the program to test. This requires a full Java grammar (which must be kept in sync with the latest updates to Java). To capture the identity of the callee, parameter values and return value of a method, one only needs to transform that particular method (i.e. locally). But inside the method there is no way to access the identity of the caller. Java does offer facilities to inspect stack frames, but these frames contain only static entities, such as the name of the method which called the currently executing method, or the type of the caller, but not the caller itself. To capture the caller, a global transformation at all call-sites is needed (and in particular one needs to have access to the source code of all clients which call the method). The same problem arises in monitoring calls to required methods. Finally it proved to quickly get very complex to handle all Java features. We wrote an initial

version of a weaver in Rascal which already took over 150 lines (over half of the full checker at the time) without supporting method calls appearing inside expressions, inheritance and dynamic binding. This approach is also unsuitable for black-box testing where only byte code is available (limiting the applicability of the tool). In summary, it is possible to implement monitoring by defining a code transformation in Rascal, but this rules out black-box testing and quickly gets complex due to the need for a full (up to date) Java grammar and the complexity of the full Java language.

The Sun debugger is part of the standard Java Development Kit, hence maintenance of the debugger is practically guaranteed. The debugger starts the original user program in a separate virtual machine which is monitored for occurences of `MethodEntryEvent` (method calls) and `MethodExitEvent` (method returns). Whenever such an event occurs the debugger can execute an event handler. However accessing the values of the parameters and return value of events is difficult, one has to use low-level `StackFrames`. As a major disadvantage, we found that the debugger is very slow (an order of magnitude slower than AspectJ), in fact it was responsible for the majority of the overhead of the run-time checker. Finally in contrast to AspectJ it not possible to add fields to objects, thus local histories never go out of scope, even if the object itself is already long destroyed.

In summary, the use of aspect-oriented programming greatly improved performance compared to the debugger-based solution and was much simpler than implementing our own weaver with code transformations, especially to handle intricate language features.

## 4 Case Study

Fredhopper provides the Fredhopper Access Server (FAS). It 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 7).

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.

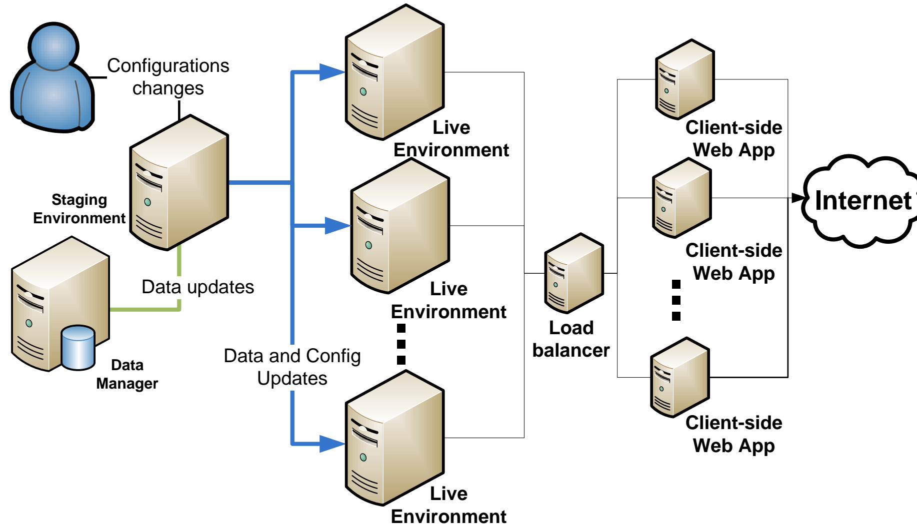



**Fig. 7.** An example FAS deployment

**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 and write accesses 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 8 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 diagram also shows a *Util* class that provides static methods for writing to and reading from `Stream`. 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*.

### 4.1 Specification

In this section we show how to modularly decompose object interaction behavior depicted by the UML sequence diagram in Figure 8 using SAGA. Figure 9 shows

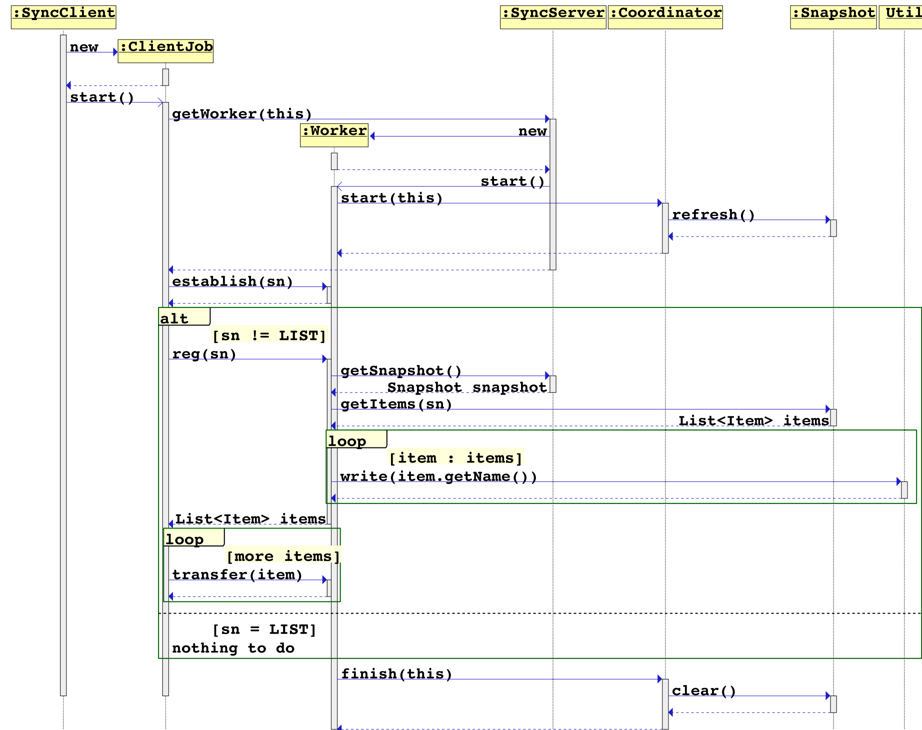



**Fig. 8.** Replication interaction

the corresponding interfaces and classes, note that we do not consider SyncClient as our interest is in object interactions of a replication session, that is after `ClientJob.start()` has been invoked.

The protocol descriptions and specifications considered in this case study have been obtained by manually examining the behavior of the existing implementation, by formalizing available informal documentations, and by consulting existing developers on intended behavior. Here we first provide such informal descriptions of the relevant object interactions:

- Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snapshot. Subsequently the invocations of methods `refresh` and `clear` should alternate.
- Coordinator: neither of methods `start` and `finish` may be invoked twice in a row with the same argument, and method `start` must be invoked before `finish` with the same argument can be invoked.
- Worker: `establish` must be called first. Furthermore `reg` may be called *if* the input argument of `establish` is not “LIST” but the name of a specific replication schedule, and that `reg` must take that name as an input argument. When the `reg` method is invoked and before the method returns, the Worker must obtain the replication items for that specific replication schedule via method `items` of the Snapshot object. The Snapshot object must be

```
interface Snapshot {
 void refresh();
 void clear();
 List<Item> items(String sn);
}

interface Worker {
 void establish(String sn);
 List<Item> reg(String sn);
 void transfer(Item item);
 SyncServer server();
}
```

```
interface SyncServer {
 Snapshot snapshot();
}

interface Coordinator {
 void start(Worker t);
 void finish(Worker t);
}

class Util {
 static void write(String s) { .. }
}
```

**Fig. 9.** Interfaces of Replication System

obtained via method `snapshot` of its SyncServer, which must be obtained via the method `server`. It must notify the name of each replication item to its interacting SyncClient. This notification behavior is implemented by the static method `write` of the class `Util`. The method `reg` also checks for the validity of each replication item and so the method must return a subset of the items provided by the method `items`. Finally `transfer` may be invoked after `reg`, one or more times, each time with a unique replication item, of type `Item`, from the list of replication items, of type `List<Item>`, returned from `reg`.

Figure 10 specifies communication views. They provide partial mappings from message types (method calls and returns) that are local to individual objects to grammar terminal symbols. Note that the specification of the Worker's behavior is modularly captured by two views: `WorkerHistory` and `WorkerRegHistory`. The view `WorkerHistory` exposes methods `establish`, `reg` and `transfer`. Using this view we would like to capture the overall valid interaction in which Worker is the callee of methods, and at the same time the view helps abstracting away the implementation detail of individual methods. The view `WorkerRegHistory`, on the other hand, captures the behavior inside `reg`. According to the informal description above, the view projects incoming method calls and returns of `reg`, outgoing method calls to `server` and `items`, and as well as the outgoing static method calls to `write`.

We now define the abstract behavior of the communication views, that is, the set of allowable sequences of interactions of objects restricted to those method calls and returns mapped in the views. Each local view also defines the file containing the attribute grammar, whoses terminal symbols the view maps method invocations and returns to. Specifically, Figure 11 shows the attribute grammars `Snapshot.g`, `Coordinator.g`, `Worker.g` and `WorkerReg.g` for views `SnapshotHistory`, `CoordinatorHistory`, `WorkerHistory` and `WorkerRegHistory` respectively.

```
local view SnapshotHistory
grammar Snapshot.g
specifies Snapshot {
 call void refresh() rf,
 call void clear() cl
}
```

```
local view CoordinatorHistory
grammar Coordinator.g
specifies Coordinator {
 call void start(Worker t) st,
 call void finish(Worker t) fn
}
```

```
local view WorkerHistory grammar Worker.g
specifies Worker {
 call void establish(String sn) et,
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 call void transfer(Item item) tr
}
```

```
local view WorkerRegHistory grammar WorkerReg.g
specifies Worker {
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 return Snapshot SyncServer.snapshot() sp,
 call List<Item> Snapshot.items(String sn) ls,
 return List<Item Snapshot.items(String sn) li,
 call static void Util.write(String s) wr
}
```

**Fig. 10.** Communication Views

The simplest grammar `Snapshot.g` specifies the interaction protocol of Snapshot. It focuses on invocations of methods `refresh` and `clear` per Snapshot object. The grammar essentially specifies the regular expression (`refresh clear`)$*$.

The grammar `Coordinator.g` specifies the interaction protocol of Coordinator. It focuses on invocations of methods `start` and `finish`, both of which take a Worker object as the input parameter. These method calls are mapped to terminal symbols $st$ and $fn$, while their inherited attribute is a `HashSet`, recording the input parameters, thereby enforcing that for each unique Worker object as an input parameter only the set of sequences of method invocations defined by the reqular expression (`start finish`)$*$ is allowed.

The grammar `Worker.g` specifies the interaction protocol of Worker It focuses on invocations and returns of methods `establish`, `reg` and `transfer`. The grammar specifies that for each Worker object, `establish` must be first invoked, then followed by `reg` and then zero or more `transfer`, that is, the regular expression (`establish reg transfer`$*$). We use the attribute definition of the grammar to ensure the following:

- The input argument of `establish` and `reg` must be the same;
- `reg` can only be invoked if the input argument of `establish` is not "LIST";

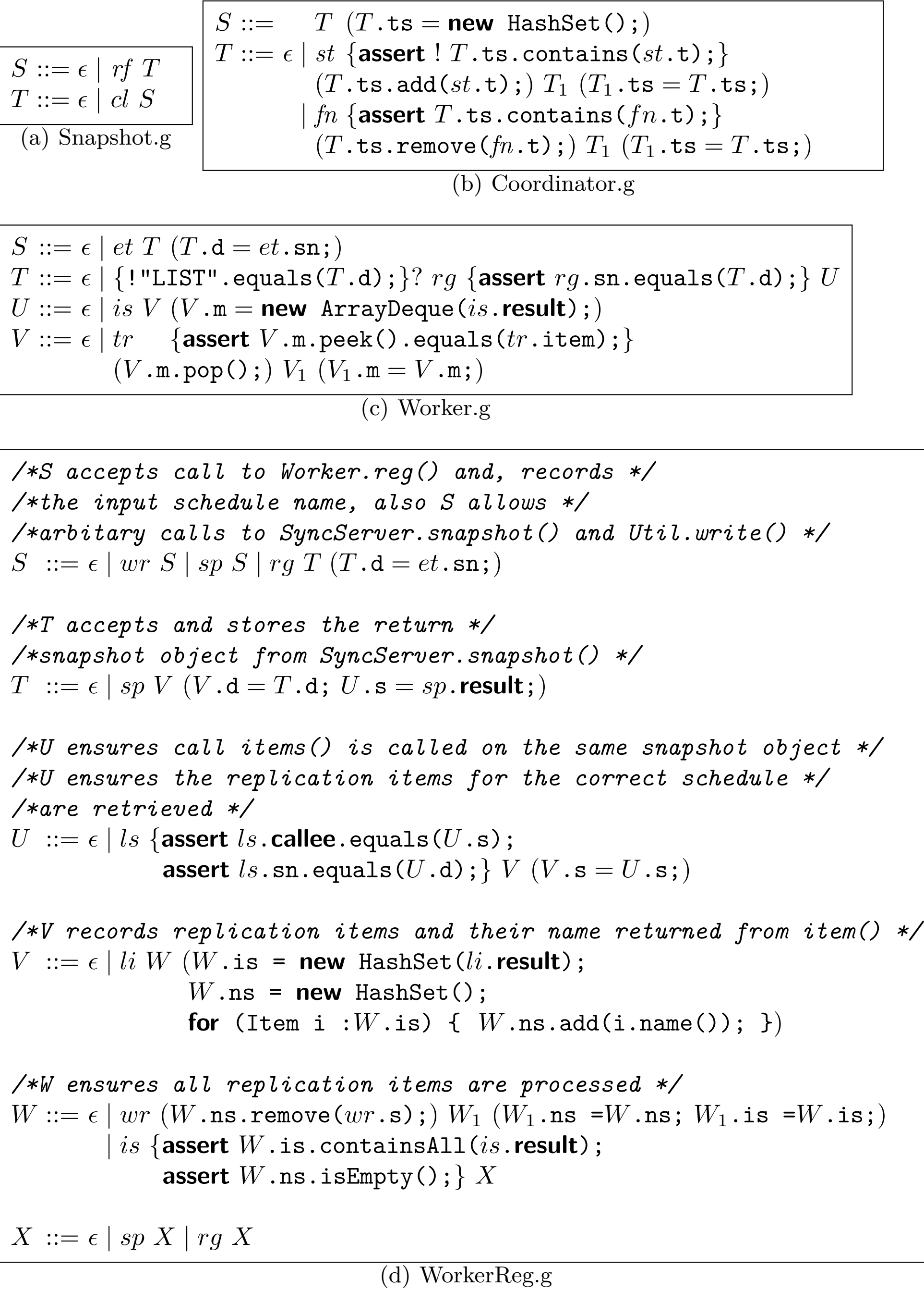



**Fig. 11.** Attribute Grammars

– The return value of reg is a list of Item objects such that transfer is invoked with each of Item in that list from position 0 to the size of that list.

The grammar WorkerReg.g specifies the behavior of the method reg of Worker. It focuses on the invocations and returns of method reg of Worker as well as the outgoing method calls and returns of Util.write and SyncServer.snapshot and Snapshot.items. At the protocol level the grammar specifies the regular expression (snapshot items write∗) inside the invocation method reg. We use attribute definition to ensure the following:

– Snapshot.items must be called with the input argument of reg and it must be called on the Snapshot object that is identical to the return value of SyncServer.snapshot;
– The static method Util.write must be invoked with the value of Item.name for each Item object in the Collection returned from Snapshot.items;
– The returned list of Item objects from reg must be a subset of that returned from Snapshot.items.

Notice that methods Util.write and SyncServer.snapshot may be invoked outside of the method reg. However, this particular behavioral property does not specify the protocol for those invocations. The grammar therefore abstracts from these invocations by allowing any number of calls to Util.write and SyncServer.snapshot before and after reg.

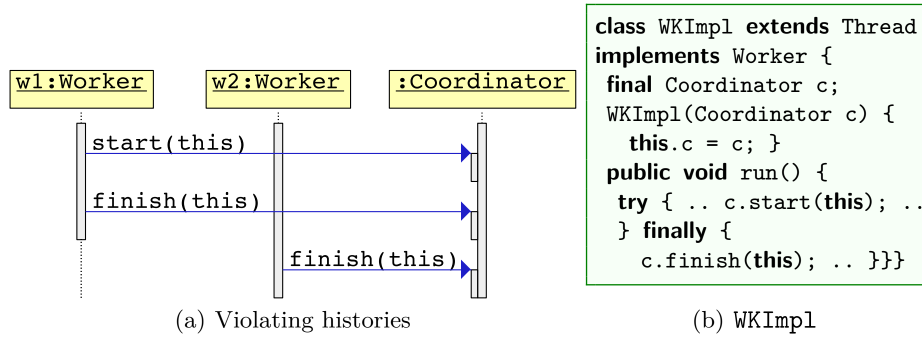
### 4.2 Experiment

We applied SAGA to the Replication System. The current Java implementation of FAS has over 150,000 lines of code, and the Replication System has approximately 6400 lines of code, 44 classes and 5 interfaces.

We have successfully integrated the SAGA into the quality assurance process at Fredhopper. The quality assurance process includes automated testing that includes automated unit, integration and system tests as well as manual acceptance tests. In particular system tests are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Functional testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and as a result we may leverage SAGA by augmenting these two automated test facilities with runtime assertion checking using SAGA.

To integrate of SAGA with the system tests, we employ Apache Maven tool[10], an open source Java based tool for managing dependencies between applications and for building dependency artifacts. Maven consists of a project object model

[10] maven.apache.org




(a) Violating histories (b) WKImpl

**Fig. 12.** Incorrect behavior

(POM), a set of standards, a project lifecycle, and an extensible dependency management and build system via plug-ins. We use its build system to automatically generate and package the parser/lexer of attribute grammars as well as aspects from views and grammars. We expose the packaged aspects, parser and lexer to FAS instance on the server farm and employ Aspectj using load-time weaver for monitoring method calls/returns during the execution of FAS instances on the server farm. Table 2 shows the number of join point matches during the execution of 766 replication sessions over live client data. Figure 13 shows the exection time of the 766 replication sessions with and without the integration of SAGA in milliseconds. Despite the fact that we cannot control the exact flow of control of the replication sessions (due to dependence on user input), the graph clearly shows that the integration of SAGA has minimal performance impact on the execution time.

| Join point | Terminal | Match |
|---|---|---|
| **call static** `write` | $wr$ | 247446 |
| **return** `snapshot` | $sp$ | 3061 |
| **call** `transferItem` | $tr$ | 1101 |
| **return** `reg` (`WorkerHistory`) | $is$ | 765 |
| **return** `reg` (`WorkerRegHistory`) | $is$ | 765 |
| **call** `establish` | $et$ | 766 |
| **call** `reg` (`WorkerHistory`) | $rg$ | 765 |
| **call** `reg` (`WorkerRegHistory`) | $rg$ | 765 |
| **return** `items` | $li$ | 765 |
| **call** `start` | $st$ | 766 |
| **call** `finish` | $fn$ | 766 |
| **call** `items` | $ls$ | 765 |
| **call** `refresh` | $rf$ | 766 |
| **call** `clear` | $cl$ | 766 |

**Table 2.** Join point matches in 766 replication sessions

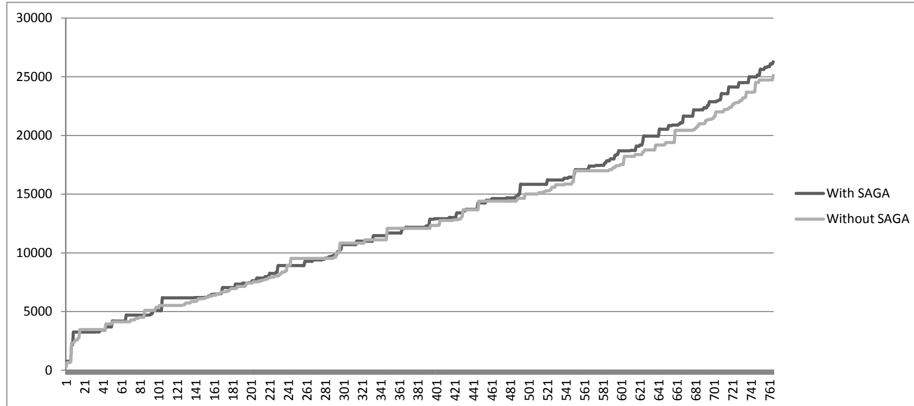



**Fig. 13.** Comparison of the execution time (milliseconds) of the replication sessions with and without the integration of SAGA

During this session we have found an assertion error at join point **call** `finish` due to the condition $T$`.ts.contains(`$fn$`.t)` not being satisfied at non-terminal $T$ of the grammar Coordinator.g. Specifically, the implementation of Worker (`WKImpl`) that invoke `finish` before `start`. Figure 12(a) shows the sequence diagram automatically generated from the output of SAGA on the invalid histories causing the assertion error. Figure 12(b) shows part of the implementation of `WKImpl`. It turns out that in the `run` method of `WKImpl`, the method `start` is invoked inside a **try** block while the method `finish` is invoked in the corresponding **finally** block. As a result when there is an exception being thrown by the execution preceding the invocation of `start` inside the **try** block, for example a network disruption, `finish` would be invoked without `start` being invoked.

## 5 Conclusion

We developed SAGA, a run-time checker which fully automatically checks properties of both the protocol behavior and data-flow of message sequences in a declarative manner. We identified the different components of SAGA and evaluated SAGA on an industrial case study of the eCommerce company Fredhopper. The results of this case study show the feasability of our method for run-time verification in industrial practice (it has already led to the integration of SAGA into the software lifecycle at Fredhopper), in contrast to methods for static verification which require both an in-depth knowledge of the case study and the underlying theorem prover. A beta version of SAGA can be found on `http://www.cwi.nl/~cdegouw`.

*Related Work* A preliminary version of a prototype of our tool containing some of the basic underlying ideas was presented at the workshop Formal Techniques

for Java-Like Programs 2010 and appeared in its *informal* proceedings[11]. In the current paper we apply and evaluate a new version to an industrial case study and succesfully integrate SAGA into the quality assurance process of Fredhopper. Based on this application and evaluation we extended our framework to support a more general class of grammars to specify data-dependent protocol behavior. Furthermore the new version features a tighter integration of attribute grammars and assertions. Finally the support for the features listed in table 1 is new.

There exist many other interesting approaches to monitoring message sequences, none of which address their integration with the general context of run-time assertion checking. Consequently all the other approaches only allow a combination of a very restricted class of data-oriented properties and protocol properties. For example, Martin et al. [15] introduce a 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. [1] develop an extension of AspectJ with a history-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. [16] monitors calls and returns with the same Java Debugger Architecture we have also evaluated in the implementation section. The debugger is very slow compared to aspect-oriented approaches. 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. [5] 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 histories. 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, comparable to built-in attributes of terminals in our setting). This limits the expression of data properties. Though to circumvent this limitation one may however hack general properties into the tool implementation. In contrast, our approach supports a general methodology to introduce systematically *user-defined* properties, by means of attributes of non-terminals. Furthermore SAGA supports conditional productions which are essential to specify protocols dependent on data in a declarative manner. Finally, JavaMOP does not directly support the specification of local histories (i.e. monitoring the messages sent and received by a single object). LARVA is developed by Colombo et al. [8]. The specification language has an imperative flavour: users define a finite state machine to define the allowed history (i.e. one has to 'implement' a regular expression themselves). It is not possible to directly express context-free protocols. Data properties are supported, though in a limited manner, by enriching the state machine with conditions on method parameters or return values. It is not

[11] Available in the ACM Digital Library with the title "Prototyping a tool environment for run-time assertion checking in JML with communication histories", authored by Frank S. de Boer, Stijn de Gouw and Jurgen Vinju

possible to specify a local history of a single object. DeLine and Fähndrich [10] propose a statically checkable typestate system for object-oriented programs. Typestate specifications of protocols correspond to finite state machines (assertions are not considered in their approach), thus for example a stack cannot be properly specified.

To the best of our knowledge, no other approach *integrates* protocol oriented properties into existing state-based assertion languages. The integration does not involve an extension of the syntax and semantics of the assertion language itself. As an important consequence, no change in the implementation of the state-based assertion checker is needed, in contrast to the following works. Cheon and Perumandla present in [6] an extension of the JML *compiler* with call sequence assertions. Call sequence assertions are regular expressions (proper context-free grammars cannot be handled) over method names and the data sent in calls and returns is not considered. Protocol properties (call sequence assertions) are handled separately from data properties, and as such are not integrated into the general context of (data) assertions. The proposed extension to call sequence assertions involves changing the existing JML-compiler (in particular, both the syntax and the semantics of JML assertions are extended), whereas in our test suite integrating with JML consists only of a simple pre-processing stage. Consequently in our approach no change in the JML-compiler is needed, and new versions of the JML-compiler are supported automatically, as long as they are backwards compatible. Hurlin [12] presents an extension of the previous work to handle multi-threading which however is not supported by run-time verification (instead it discusses static verification). As in the previous work, an integration of protocol properties with assertions is not considered. Trentelman and Huisman [21] describe a new formalism extending JML assertions with Temporal Logic operators. A translation for a subset of the Temporal Logic formulae back to standard JML is described, and as future work they intend to integrate their extension into the standard JML-grammar which requires a corresponding new compiler.

*Future Work* SAGA visualizes the offending history of a Java program that violates the given attribute grammar in the form of a UML sequence diagram. For industrial applications the histories (and consequently the corresponding diagram) can get very large, even when projecting away irrelevant events with the communication view. In such cases we found it is very useful to (further) filter events from the diagram, focussing on a specific part of the diagram. For instance, only showing all events in which a particular object was involved. The sequence diagram editor used by SAGA provides preliminary support for filtering using low-level UNIX system-based utilities grep and sed, but more high-level solutions specifically taylored for sequence diagrams would be even more useful. Furthermore for debug purposes it would be convenient to visualize the current contents of the heap.

Another line of future work concerns off-line monitoring. Off-line monitoring serializes and stores the history of a running program in a file. This file is checked later for correctness, possibly on a different computer. This allows companies to

enable monitoring production code deployed at clients with little performance penalty: the histories can be checked on dedicated computers at the company instead of at the client. A potential disadvantage of off-line monitoring is that it is not possible anymore to stop a running system directly after the attribute grammar is violated (or inspect the content of the heap at that time).

*Acknowledgements* We wish to express our gratitude to Behrooz Nobakht for his help on the integration with the Java debugger and Jurgen Vinju for the helpful discussions and major contributions to our Rascal tool.

## References

1. 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.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.
3. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
4. P. Chalin, P. R. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In *VSTTE*, pages 70–83, 2008.
5. F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.
6. Y. Cheon and A. Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, 2007.
7. 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.
8. C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *SEFM*, pages 33–37, 2009.
9. S. de Gouw and F. S. de Boer. Run-time verification of black-box components using behavioral specifications: An experience report on tool development. In *FACS*, 2012.
10. R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
11. G. Hedin. Incremental attribute evaluation with side-effects. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation, 2nd CCHSC Workshop, Berlin GDR, October 10-14, 1988, Proceedings*, volume 371 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 1988.
12. C. Hurlin. Specifying and checking protocols of multithreaded classes. In *ACM Symposium on Applied Computing (SAC'09)*, pages 587–592. ACM Press, 2009.
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. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPLSLA*, 2005.
16. 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.
17. T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.
18. T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *In Computational Complexity*, pages 263–277. Springer-Verlag, 1994.
19. A. Pnueli and A. Zaks. Psl model checking and run-time verification via testers. In *FM*, pages 573–586, 2006.
20. M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.
21. K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *AMAST*, pages 334–348, 2002.