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

Stijn de Gouw
CWI, Amsterdam, The Netherlands
Leiden University, The Netherlands
cdegouw@cwi.nl

Frank S. de Boer
CWI, Amsterdam, The Netherlands
Leiden University, The Netherlands
frb@cwi.nl

Peter Y. H. Wong
Fredhopper B.V., Amsterdam, The Netherlands
peter.wong@fredhopper.com

Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

ABSTRACT

We introduce SAGA, a general framework that *combines* monitoring and run-time assertion checking. SAGA integrates both data-flow and control flow properties of Java classes *and interfaces* in a single formalism. We evaluate the framework by conducting an industrial case study.

Categories and Subject Descriptors

D2.1 [**SOFTWARE ENGINEERING**]: Requirements-/Specifications—*Tools*; D2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Tracing*

Keywords

Histories, Traces, Attribute Grammar, Case Study, Specification, Run-time Assertion Checking, Run-time Verification, Evaluation

1. INTRODUCTION

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. Compared to program logics, run-time verification emphasizes *executable specifications*. Further, whereas program logics statically cover all possible execution paths, which is generally undecidable, run-time verification is a fully automated, on-demand validation process which applies to the actual runs of the program.

Run-time verifiers can be divided in two general categories: run-time assertions checkers, which specify the data-flow of a program, and monitors, which specify the control-flow of a 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 by their very nature can neither be used to specify the *interaction protocol* between objects, which is in contrast to other formalisms such as message sequence charts and UML sequence diagrams, nor can state-based assertions be used to specify interfaces since interfaces do not have a state[1]. On the other hand, there exists many monitoring tools (MOP, PQL, Larva, Tracematches) which specify and check control-flow (or protocol-oriented) properties, but do not specify the data-flow: MOP, PQL, Larva, Tracematches, JmSeq, UTJML.

The main contribution of this paper is twofold. Firstly, we introduce SAGA, a general framework that *combines* monitoring and run-time assertion checking. In contrast to all of the above tools, SAGA integrates both data-flow and control flow properties of Java classes *and interfaces* in a single formalism. Table 1 lists the main features supported by SAGA. Secondly, we provide an evaluation on the expressiveness and the usability of the current state of the art tools for run-time assertion checking by conducting an industrial case study from the eCommerce software company Fredhopper.

The basic idea underlying our framework 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* [9] extended by assertions. Attribute grammars

*This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu/`)

SAC'13 March 18-22, 2013, Coimbra, Portugal.

[1] 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).

allow the high-level specification of user-defined abstractions of message sequences (e.g., their length) in terms of the attributes of the grammars describing these sequences. 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.

Constructors
Inheritance
Dynamic Binding
Overloading
Static Methods
Required Methods
Access Modifiers

Table 1: Supported features

Related Work.

This paper extends a previous workshop paper [5] as follows: It significantly extended the tool to a much larger part of Java (all features in table 1 are new, and more). The specification language is much more expressive by including assertions and conditional productions in attribute grammars. We applied and evaluated our tool on an industrial case study.

There exist many other interesting approaches to monitoring message sequences which however (as already remarked above) do not address their integration with the general context of run-time assertion checking. The experience report in Section 5 contains an in-depth comparison with those.

Cheon and Perumandla present UTJML in [3] 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 [7] 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 [13] 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.

2. CASE STUDY

Fredhopper provides the Fredhopper Access Server (FAS). It is a distributed concurrent object-oriented system that provides search and merchandising services to e-Commerce 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(a)).

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.

Replication Protocol

The SyncServer communicates to SyncClients by creating *Worker* objects. Workers serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering the read/write access of the staging environment's underlying file system. The SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 1(b) depicts a UML sequence diagram concerning parts of the replication protocol with the interaction between a ClientJob, a Worker, a Coordinator and a Snapshot. The figure assumes that a SyncClient has already established connection with a SyncServer and that both a ClientJob from the SyncClient and a Worker from a SyncServer have been instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a session.

3. THE MODELING FRAMEWORK

Objects are not static parts of a system, but evolve through interaction with their environment. Abstracting from the implementation details, an execution of an object can be represented by its *communication history*, i.e., the sequence of messages corresponding to the invocations and completions of its methods (as declared by its interface).

In this section we describe our modeling framework in Java for the behavioral description of an object interface, expressed in terms of its communication histories. We extend attribute grammars with assertions to specify properties of user-defined abstractions of communication histories. We explain the basic modeling concepts by formalizing three different properties of the interfaces shown in Figures 2(a) to 2(c) from the case study, each focusing on a different behavioral aspect.

Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snap-

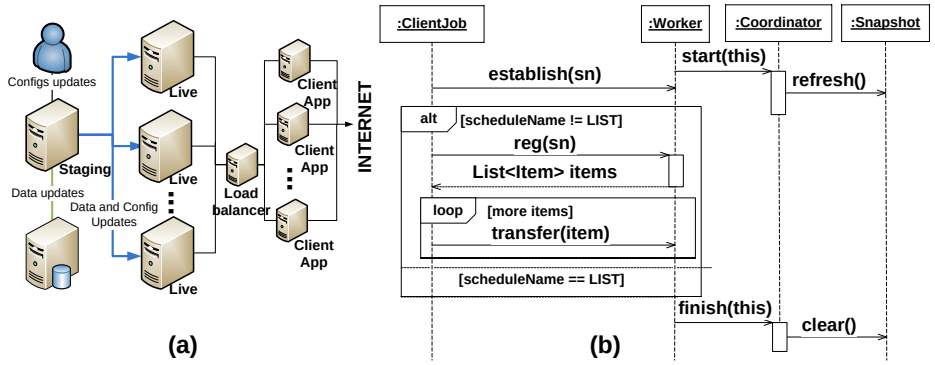

Figure 1: (a) An example FAS deployment and (b) Replication interaction

shot. Subsequently the invocations of methods `refresh` and `clear` should alternate. *Coordinator*: neither of the methods may be called twice in a row with the same argument, and method `start` must be called before `finish`. *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. Finally `transfer` may be called 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`.

```
interface Snapshot {
 void refresh();
 void clear();
}
```

(a) `Snapshot`

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

(b) `Coordinator`

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

(c) `Worker`

Figure 2: Interfaces of Replication System

Consider an instance of a class implementing the `Coordinator` interface. The messages in the history of this object are modeled in our framework as instances of the *message types*: **call**-`start(Worker t)`, **return**-`start(Worker t)`, **call**-`finish(Worker t)` and **return**-`finish(Worker t)`. These message types uniquely identify invocations and completions of the methods `start` and `finish`. Note that message types distinguish between overloaded methods by taking into account the parameter types of the method in question. The return type is not needed to distinguish methods since Java does not allow overloading on return type.

In general, for every method signature `T m(`T_1 $u_1, \ldots$`)` the modeling framework defines message types **call**-`m(`T_1 $u_1, \ldots$`)` and **return**-`m(`T_1 $u_1, \ldots$`)`. Each call will be represented in the communication history by an object which stores the actual parameters and each return is represented by an object storing the return value. Henceforth we call the classes of such objects token classes.

A *communication view* is a partial mapping from message types to grammar terminal symbols. Communication events of an unmapped message type are projected away. Naming the relevant events allows the user to use intuitive names for the selected messages and enables identifying two distinct messages by the same name (this is not used in the examples shown here). For example, the communication view in Figure 3(b) introduces an abstraction of the communication history in terms of its *projection* onto the messages which correspond to invocations of the `start` and `finish` methods, using the names st and fn. We thus abstract in this particular case from the returns of these methods. Note that the communication views in Figures 3(a) to 3(c) omit the types of the parameters. This is possible because the interfaces shown above did not contain any overloaded methods, hence each message type can be identified unambiguously even when type information is omitted. In general SAGA supports multiple communication views for a given interface which allow the developer to focus on the different behavioral aspects of the interface.

```
view SnapshotProc {
 call-refresh rf,
 call-clear cl
}
```

(a) `Snapshot`

```
view CoordinatorProc {
 call-start st,
 call-finish fn
}
```

(b) `Coordinator`

```
view WorkerProc {
 call-establish et,
 call-reg rg,
 return-reg is,
 call-transfer tr }
```

(c) `Worker`

Figure 3: Communication Views

The abstract behavior of a communication view can be defined in terms of sets of sequences of the names introduced in the view (i.e. sets of histories). Attribute grammars provide a powerful and high-level way to define such sets. The names for the messages specified in the communication view form the *terminals* of the grammar.

Figure 4(a) shows the property of the 'Snapshot' interface. The context-free grammar describes the prefix closure of sequences of the terminals 'refresh' and 'clear' as given by the regular expression (refresh clear)*. As the property

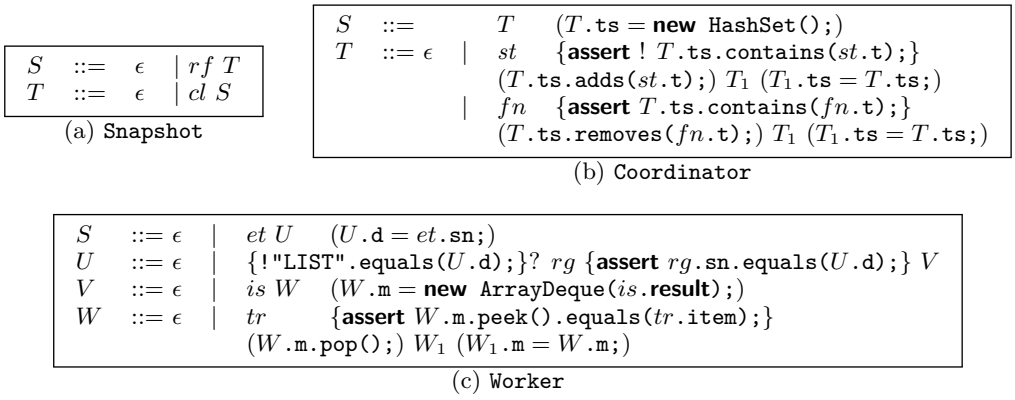

Figure 4: Attribute Grammars

does depend on data, there are no attributes in the grammar. Parse errors correspond to violations of the protocol as defined by the grammar. Note that in general the specification of the *ongoing* behavior of an object requires prefix closed grammars. Furthermore, it is important to observe that a grammar describes the protocol behavior of a single object.

We now turn to the second property described informally in the beginning of this section, which features attributes. In each attribute grammar, terminals have *built-in* attributes given by their message type as defined in the communication view, whereas non-terminals have *user-defined* attributes as given in the grammar. More specifically, built-in attributes for terminals corresponding to a message type '**call-m**' store the values of the actual parameters, and terminals corresponding to a '**return-m**' message type have a single built-in attribute '**result**' (if `m` does not have return type **void**) storing the return value. Actual parameters can be accessed in the grammar by the names of their corresponding formal parameters given in the interface.

The grammar in Figure 4(b) formalizes the Coordinator property. Attribute definitions are written between (), whereas assertions over attributes are written between {}. We define an inherited attribute 'ts' of the non-terminal T to record the value of built-in attribute (in this case, a method parameter) 't' from terminals 'st' and 'fn'. Once the attribute 'ts' is defined by setting T_1`.ts =`T`.ts;`, an assertion (either **assert** `T.ts.contains(st.t);` or **assert** T .ts.contains(st.t);!, depending on the chosen production) checks the desired property on the data. Assertions in a grammar production can be written at any position in a production rule and are evaluated during parsing at that position. As an example, the assertion on the second line is evaluated directly after 'st' is parsed, but before `T.ts.adds(st.t);` executes and T_1 is parsed. Assertions used in grammars specify data properties of parts of the history. As a special case, assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions can be asserted directly after the terminal.

The grammar in Figure 4(c) formalizes the last property described in the beginning of the section. The non-terminal U has an inherited attribute 'd' of type `String`, and the non-terminal W has an inherited attribute 'm' of type `ArrayDeque` (a Java implementation of a stack), which record input arguments and return values of method calls respectively. However even in combination with assertions this is not enough to obtain a faithful formalization of the described property. In particular, since `reg` may be called only depending on the value of the input argument of `establish`, this particular protocol depends on data. We therefore consider attribute grammars enriched by *conditional productions* [12]. In such an extended grammar, a production is chosen only when the given condition (a **boolean** expression over the attributes) for that production is true, hence conditions are evaluated before any of the symbols in the production are parsed, and before attributes are set and assertions are evaluated. The first rule for the non-terminal U is a conditional production which ensures that whenever method `reg` is called, the input argument must be the name of the replication schedule received in the method call `establish`. Note that in contrast to assertions, conditions in productions affect the parsing process.

In summary, a communication view introduces a user-defined abstraction of communication histories in terms of the declaration of the terminals of the attribute grammar. The given interface provides a name-space of the message types of these terminals. The rules of the grammar define *invariant* properties of the high-level protocol structure of the corresponding abstraction. Assertions in the grammar are introduced to specify data-oriented properties of (parts of) the communication history.

4. TOOL SUPPORT

Tool support is provided by SAGA, a 600 line meta-program written in Rascal (a powerful meta-programming language). SAGA has a component-based design, combining a parser generator, a state-based assertion checker and a monitoring tool for Java programs. Each of those components is instantiated by a state-of-the-art tool, discussed in the remainder of this section. An overview of the tool architecture is shown in Figure 5.

ANTLR [11] is a popular **parser generator** which generates a recursive descent Java parser for a given attribute grammar. Attributes are defined using semantic actions (a Java statement executed whenever some production rule is chosen during parsing) and there is support for streams of custom token classes. ANTLR also supports conditional productions (semantic predicates in ANTLR terminology), which allow even certain context-sensitive grammars to be parsed. Alas, it does not support general context-free grammars (in particular, left recursive grammars are unsupported) or incremental parsing. Incremental parsing allows reusing

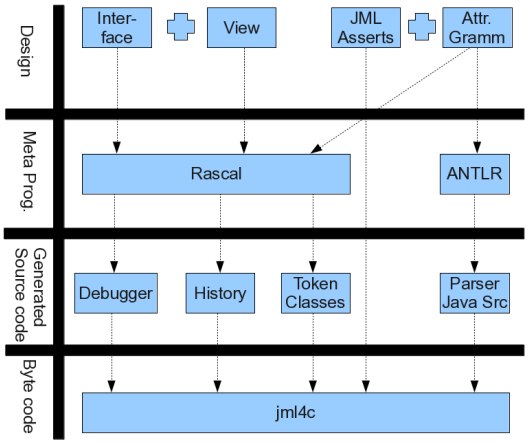

Figure 5: Tool architecture of SAGA

(parts of) the parse tree of a prefix in parsing the whole string which is of major importance for optimization purposes. We have not been able to find any Java parser generator which supports general context-free grammars and incremental parsing. The assertions occuring in the grammar are evaluated by the standard java compiler, which serves as the **state-based assertion checker**.

We investigated two alternatives for the **monitoring** component: the Sun JDI Debugger and AspectJ. The Debugger creates a wrapper for the main class of the Java program under test. The wrapper starts the original program inside a new virtual machine and monitors any method calls or returns executed on this virtual machine (i.e. it enables tracing). The debugger does not modify the source code of the original program and automatically ensures that the semantic actions in the attribute grammar do not affect the state of the program under test (since the program under test is executed in a separate virtual machine). In contrast, AspectJ modifies the source code (or bytecode, the actual Java source code does not need to be available) of the original program to intercept method calls or returns. Care must be taken to write semantic actions in the attribute grammar which do not have side-effects, since such actions modify the state of the program under test. In principle both the Debugger and AspectJ suffice for our purposes, but AspectJ is almost an order of magnitude faster than the debugger. We have therefore chosen to use AspectJ in the implementation of SAGA.

Rascal [8] is a domain specific language for meta programming. SAGA uses its parsing, source code analysis, source-to-source transformation and source code generation features to instantiate the modeling framework. In particular SAGA generates 'token classes' for each message type. The fields of a token class are the formal parameters, a `caller` and `callee` field and (only for return messages types), a field `result`. SAGA further generates Java source code for a History class, which represents the current history of the program as a list of token classes. Finally SAGA generates AspectJ code to intercept method calls and returns, and update the history accordingly. Whenever the history is updated, the parser (generated by ANTLR) is triggered by the history class to parse the new history, and compute the corresponding new attribute values.

5. EXPERIENCE REPORT

We applied SAGA to the Replication System, part of the Fredhopper Access Server (FAS). 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. While we used standard Java assertions in our case study, we have also experimented on integrating SAGA with the run-time assertion checking facilities provided by OpenJML[2], which is one of the more actively developed JML implementations. This has resulted in many valuable improvements to the development of OpenJML. Specifically the OpenJML compiler have had issues with type checking **synchronized** blocks and **enum** types as well as parsing and compiling the source code of FAS. See `http://sourceforge.net/tracker/?group_id=65346&atid=510629` for the kind of issues we have encountered when using OpenJML. Due to these issues we have decided not to pursue integration with JML in our case study. Nevertheless, most of the issues reported have been resolved in the latest version of OpenJML.

We now proceed with a direct comparison of SAGA, PQL [10], Jassda [1], LARVA [4] and MOP [2].

	`Snapshot`	`Coordinator`	`Worker`
PQL	yes	no	no
Jassda	yes	no	no
LARVA	yes	yes	yes
MOP	yes	yes	yes
SAGA	yes	yes	yes

Table 2: Comparison of Expressiveness

We investigated the expressiveness of the specification languages of these tools by attempting to express and check the SnapShot, Worker and Coordination properties (see Section 3). Due to space limitations we have put the resulting specifications in the full version [6]. Table 2 summarizes the results. Neither PQL nor Jassda can express the `Coordinator` and `Worker` properties since neither allows user-defined properties of data. LARVA and MOP, on the other hand, support executing arbitrary Java statements when an event occurs, hence it is possible to define data-oriented properties such as `Coordinator` and `Worker`. As such, user-defined properties of the data of a single event are possible to express. It is not possible to directly express properties of *sequences* of events (i.e. the data-flow of the history). In LARVA, non-regular context-free protocols cannot be expressed *directly*: one would have to write the parser for a context-free grammar oneself. The user would then essentially be writing their own run-time checker in Java, bypassing MOP and Larva. This is clearly unfeasible, and the resulting specifications are not declarative anymore. Most importantly, in that degenerative sense of expressiveness, AspectJ (on which MOP and LARVA are based) would already be sufficient. Learnability is the capability of a software product to enable the user to learn how to use it. Table 3 shows the number of hours spent on activities to specify and monitoring properties defined in Figure 4.

The most time spent at specification was for PQL; PQL defines a new specification language for expressing queries for (recursively) matching sequences of method invocations. We find the language to be counter-intuitive as it does not

[2]`http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`

	Documentation	Maintenance	Support
PQL	1 paper, examples	2006	Minimal
Jassda	papers, (German) thesis, examples	2006	Minimal
LARVA	papers, manuals, examples	2011	Immediate
MOP	papers, manuals, examples	2011	Immediate
SAGA	papers, examples	2012	Immediate

Table 4: Adoptability

	Specification	Execution
PQL	5	2
Jassda	4	2
LARVA	2	1
MOP	5	1
SAGA	3	1

Table 3: Duration per Activity in hours

match any existing modeling or programming languages. Moreover, it requires the user to specify *invalid* behavior rather than valid ones and it is unclear how to specify method invocations with specific input values. Similarly Jassda lacks an integration into the general context of assertion checking, which is needed to specify properties of variable values. LARVA provides an intuitive language for specifying regular protocols. Specifications are finite state automata with optionally actions (arbitrary Java code) on the transitions of the automaton. Actions can be used to express data-oriented properties, though in an imperative style. Context-free protocols are however much more cumbersome to express as noted previously. Despite the fact that the Worker property has only been formalized partially in LARVA due to requirements to express *all invalid* sequences of method invocations, the full specification in SAGA is much more concise. Though it is no so difficult in MOP to formalize the protocol behavior of the Worker [6] (data-oriented properties are more problematic, as these cannot be expressed directly as mentioned), the meaning of the grammars in MOP is unclear: the failure handler was triggered by MOP even for correct programs. Whether this is due to misunderstanding on our part of the meaning of MOP specifications, or due to a bug in MOP remains unclear even after a thorough reading of the documentation. For PQL, most time is spent identifying which Java statements are supported and how variables can be manipulated. The actual set-up of the run-time checking (compilation, instrumentation etc.) are carried by mirroring the setting in the toy examples provided by the installation package. For Jassda, time is spent at understanding the Java Debugger Architecture, and in particular the proper settings in the configuration files.

We evaluated how easily the frameworks can be **adopted** or integrated into the the software development cycle in an industrial context such as at Fredhopper. This includes operational steps like installation, execution, and documentation and support. The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. This type of testing requires a running FAS instance and can be augmented with run-time assertion checking techniques. Lack of support and maintenance (Table 4) reduces the confidence in PQL and Jassda.

6. CONCLUSION

We developed a general modeling framework SAGA which seamlessly integrates attribute grammars for the specification of user-defined abstractions of message sequences into state-based assertion languages like JML. Our approach allows a natural way to use assertions to specify declaratively high-level data-oriented properties of these user-defined abstractions. We discussed the corresponding tool-support based on a generative framework for run-time assertion checking in Java and its application to an industrial case study. The promising results of this case study provide a solid basis for a further integration of SAGA into the software lifecycle at Fredhopper.

7. REFERENCES

[1] M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the jdi. *Electr. Notes Theor. Comput. Sci.*, 70(4), 2002.

[2] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[3] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of java programs. *Software Quality Journal*, 15(1):7–25, 2007.

[4] C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *SEFM*, pages 33–37, 2009.

[5] F. S. de Boer, S. de Gouw, and J. Vinju. Prototyping a tool environment for run-time assertion checking in jml with communication histories. FTFJP '10, pages 6:1–6:7, New York, NY, USA, 2010. ACM.

[6] S. de Gouw. Full paper version and tool download (`http://www.cwi.nl/~cdegouw`), 2012.

[7] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *ACM Symposium on Applied Computing (SAC'09)*, pages 587–592. ACM Press, 2009.

[8] 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, *SCAM 2009*, pages 168–177, 2009.

[9] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[10] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPLSLA*, 2005.

[11] T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[12] 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.

[13] K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *AMAST*, pages 334–348, 2002.