Run-Time Verification of Black-Box Components using Behavioral Specifications: An Experience Report on Tool Development *

Frank S. de Boer[1,2] and Stijn de Gouw[1,2]

[1] CWI, Amsterdam, The Netherlands
[2] Leiden University, The Netherlands

Abstract. We introduce a generic component-based design of a run-time checker, identify its components and their requirements, and evaluate existing state of the art tools instantiating each component.

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 [3]. Compared to program logics, assertion checking emphasizes *executable specifications*. Whereas program logics statically cover all possible execution paths, run-time assertion checking is fully automated, and applies on demand to the actual runs of the program.

By their very nature, assertions are state-based in that they describe properties of the program variables (fields of classes and local variables of methods). In general, assertions expressed in languages supporting design by contract (like the Java Modeling Language (JML) [1]) cannot be used to specify the *interaction protocol* between objects or components, in contrast to other formalisms such as message sequence charts and UML sequence diagrams. Nor can state-based assertions be used to specify component interfaces since such interfaces do not have a state[3].

This paper reports on an integrated tool environment which provides a smooth integration of the specification and run-time checking of both data- and protocol-oriented properties of component interfaces. The basic idea underlying our framework is the representation of message sequences as words of a language generated by a grammar. The formalism of *attribute grammars* allows the high-level specification of user-defined abstractions of message sequences in terms of attributes of grammars describing these sequences. We introduce a generic

* 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/`)

[3] 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).

component-based design which supports run-time checking of assertions about these attributes, which involves parsing the generated sequences of messages. We identify the components and their requirements, and evaluate existing state of the art tools which instantiate the components of the generic tool architecture.

Related Work. A preliminary version describing a prototype of an instantiation of our tool architecture was presented at the workshop "Formal Techniques for Java-Like Programs 2010" and appeared in its *informal* proceedings[4]. This prototype was based on state of the art tools. However, for industrial usage we need a component-based design (as described above), and an experience report on various instantiations of the generic tool design.

There exist many other interesting approaches to run-time verification and monitoring of message sequences which however do not address its integration with the general context of run-time assertion checking, e.g. JML: PQL, Tracematches, JmSeq, LARVA, Jass and JavaMOP. Due to space limitations we do not further discuss these approaches individually.

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 will use the interface of the Java `BufferedReader` (Figure 1) as a running example to explain the basic modeling concepts.

```
interface BufferedReader {
  void close();
  int read();
}
```

Fig. 1. Relevant methods of the BufferedReader Interface

```
S ::= open C1   assert open.caller != null
                     ==> open.caller == C1.caller;
    | ϵ
C ::= read C1   C.caller = C1.caller;
    | close S1  C.caller = close.caller;
    | ϵ         C.caller = null;
```

Fig. 2. Extended Attribute Grammar modeling the behavior of a BufferedReader

To each method `m` in the interface we associate two *communication events*: 'call-m' and 'return-m'. The *observable* communication history of an object of a

[4] 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

class implementing the above interface consists of sequences of communication events.

Context-free grammars provide a declarative way to define the allowed histories of an object. The context-free grammar underlying the attribute grammar in Figure 2 generates the valid histories for `BufferedReader`, describing the *prefix closure* of sequences of the terminals call-BufferedReader', 'call-read' and 'call-close' as given by the regular expression (call-BufferedReader call-read* call-close). Note that since grammars specify *invariant* properties of the ongoing behavior of an object, they must be prefix-closed. In general, communication events form the terminal symbols of the grammar, and non-terminal symbols specify the valid sequences of communication events.

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. A terminal symbol 'call-m' has *built-in* attributes 'caller', 'callee' and the parameter names for respectively the actual parameters and object identities of the caller and callee. A terminal 'return-m' additionally has an attribute `result` referring to the return value. Non-terminals have *user-defined* attributes to define data properties of sequences of events. 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 2 a user-defined attribute 'caller' for the non-terminal 'C' is defined storing the identity of the object which closed the `BufferedReader` (and is `null` if the reader was not closed yet). The assertion allows only those histories in which the object which opened (created) the reader also closed it.

Assertions can be placed at any position in a production rule and are evaluated there. Note that assertions appearing directly before a terminal can be seen as a precondition of a 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 followed by a different assertion. Hence different preconditions (or postconditions) can be used for the same method, depending on the context (grammar production) in which the call was made.

3 Generic Tool Architecture

Given a Java interface specified with an attribute grammar, we would like to test whether an object implementing the interface satisfies the properties defined in the grammar at every point in its lifetime. In this section we describe a generic tool architecture which achieves this. Four different components are combined: a state-based assertion checker, a parser generator, a debugger and a general tool for meta-programming. Traditionally these tools are used for very diverse

purposes and don't need to interact with each other. We therefore investigate requirements needed to achieve a seamless integration of these components, motivated by describing the workflow of the run-time checker.

Suppose that during execution of a Java program, a method of a class (subsequently referred to as CUT, the 'class under test') which implements an interface specified by an attribute grammar is called. The new history of the object on which the method was called should be updated to reflect the addition of the method call. To represent the history of an object of CUT, the **Meta-Programming** tool generates for each method `m` in CUT two classes `call-m` and `return-m`. These classes contain the following fields: the object identitity of the *callee*, the identity of the *caller* and the actual parameters. Additionally `return-m` contains a field `result` containing the return value. A Java `List` containing instances of `call-m` and `return-m` then stores the history of an object of CUT.

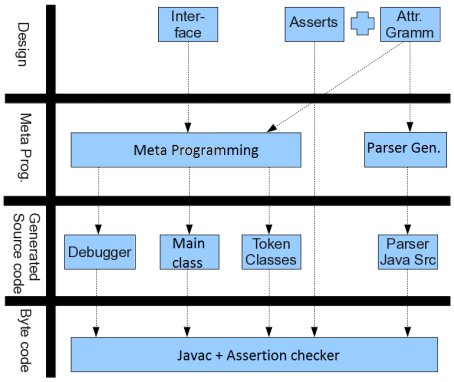

Fig. 3. Generic Tool Architecture

The meta-programming tool further generates code for a wrapper class which replaces the original main class. This wrapper class contains a field `H`, a Java `map` containing pairs (`id`, `h`) of an object identity `id` and its local history `h`. The new main class executes the original program inside the **Debugger**. The Debugger is responsible for monitoring execution of the program. It must be capable of temporarily 'pausing' the program whenever a call or return occurs, and execute user-defined code to update `H` appropriately . Moreover the Debugger must be able to read the identity of the callee, caller and parameters/return-value.

After the history is updated the run-time checker 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 heterogenous list of `call-m` and `return-m` objects, the parser must support parsing streams of tokens with user-defined types. Assertions in general describe properties of Java objects, and the grammar contains assertions over attributes, the attributes must be normal Java variables. Consequently the parser generator must allow arbitrary user-defined java code (to set the attribute value) in rule actions. The use of Java code ensures the attribute values are computable. Since assertions are allowed in-between any

two (non)-terminals, the parser generator should support user-defined actions between arbitrary grammar symbols. At run-time, the parser is triggered whenever the history of an object is updated. The result is either a parse error, which indicates that the current communication history has violated the protocol structure specified by the attribute grammar, or a parse tree with new attribute values. During parsing, the **Assertion Checker** evaluates the assertions in the grammar on the newly computed attribute values. To avoid parsing the whole history of a given object each time a new call or return is appended, ideally the parser should support incremental parsing [4]. An incremental parser computes a parse tree for the new history based on the parse trees for prefixes of the history. In our setting, the attribute grammar specifies invariant properties of the ongoing behavior. Hence the parser constructs a new parse tree after each call/return, consequently parse trees for all prefixes of the current history can be exploited for incremental parsing.

4 Instantiating the Generic Tool Architecture

The previous section introduced the generic tool architecture, which was based on four different components: meta-programming, debugger, parser generator and state-based run-time assertion checker. Here we instantiate these four components with particular (state of the art) tools, and report our experiences.

Rascal [5] is a powerful tool-supported meta-programming language tailored for program analysis, program transformation and code generation. We wrote a Rascal program of approximately 600 lines in total which generates the classes `call-m`, `return-m`, the new main class, and glue code to trigger the debugger and parser. Rascal is still in an alpha stage, it is not fully backwards compatible and we discovered numerous bugs in Rascal during development of the Rascal program. However overall our experience was quite positive. All bugs were fixed quickly by the Rascal team, and its powerful parsing, pattern matching and transforming concrete syntax features proved indispensable.

We evaluated Sun's implementation of the Java Debugging Interface for the debugger component. It is part of the standard Java Development Kit, hence maintenance of the debugger is practically guaranteed. The Sun debugger starts the original user program in a virtual machine which is monitored for occurences of `MethodEntryEvent` (method calls) and `MethodExitEvent` (method returns). It allows defining event handlers which are executed whenever such events occur. It also allows retrieving the caller, callee, parameters values and return value of events using `StackFrames`. The Sun debugger meets all requirements for the debugger stated above. As the main disadvantage, we found that the current implementation of the debugger is very slow. In fact it was responsible for the majority of the overhead of the run-time checker. This is not necessarily problematic: as testing is done during development, the debugger will typically not be present in performance critical production code. Moreover, one usually wants to test only up to a certain bound (for instance, in time, or in the number of events), and report on results once the bound is exceeded. Nonetheless, for testing up to huge bounds, a different implementation for the debugger is needed.

We instantiated the parser generator component with ANTLR, a state of the art parser generator. It generates fast recursive descent parsers for Java and allows grammar actions and custom token streams. It even supports *conditional productions*: such productions are only chosen during parsing whenever an associated Boolean expression (the condition) is true. Attribute grammars with conditional productions express protocols that depend on data which are typically not context-free. ANTLR can only handle LL(*) grammars[5], and it lacks support for incremental parsing, though this is planned by the ANTLR developers. We could not find any Java parser generator which supports general context-free grammars and incremental parsing of attribute grammars.

We 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 instance 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 proprietary compiler each time the standard compiler is updated. This problem is recognized by the JML developers [2]. OpenJML, 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.

A (variant of) the above tool suite can be obtained from `http://www.cwi.nl/~cdegouw`. It was applied successfully to an industrial size case study of the eCommerce software company Fredhopper.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.

[5] A strict subset of the context-free grammars. Left-recursive grammars are not LL(*).