Universiteit
Leiden
The Netherlands

Combining Monitoring with Run-time Assertion Checking

Gouw, C.P.T. de

Citation
Gouw, C. P. T. de. (2013, December 18). *Combining Monitoring with Run-time Assertion Checking*. Retrieved from https://hdl.handle.net/1887/22891

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/22891

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/22891 holds various files of this Leiden University dissertation

Author: Gouw, Stijn de
Title: Combining monitoring with run-time assertion checking
Issue Date: 2013-12-18

Implementation 4

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 chapter we first describe the generic architecture of our tool SAGA [34] 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. In the next section we instantiate the four components with concrete state-of-the-art tools.

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 con-

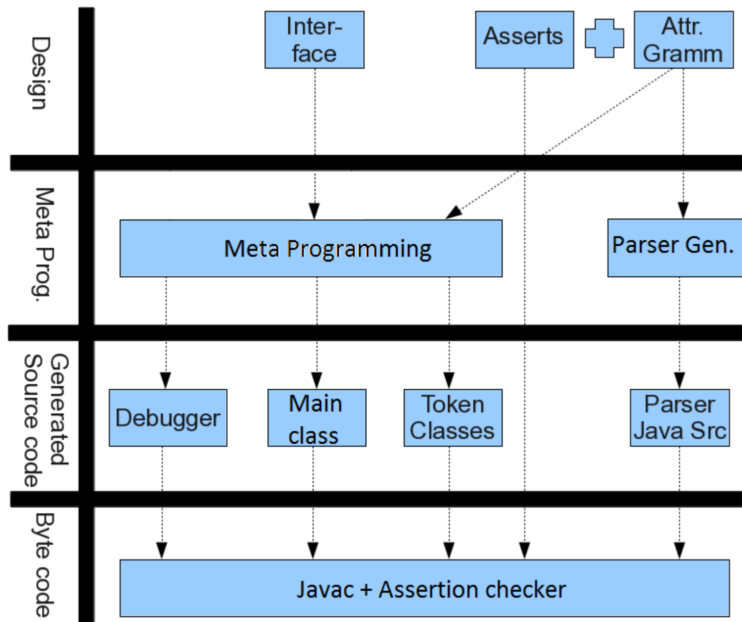

Figure 4.1: Generic Tool Architecture

tain 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.

The meta-programming tool further generates code for a wrapper class which replaces the original main class. We will refer to this class as the "history class". This history class contains a field `H`, a Java `map` containing pairs (`id`, `h`) of an object identity `id` and its local history `h`. Moreover it stores the current values of the synthesized attributes of the start symbol, these can be used in assertion languages supporting design by contract (See Section 5.1 for an example of this usage). The history 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 [43]. 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 in-

cremental parsing.

To illustrate how the tools described above interact with each other at run-time, the UML sequence diagram in Figure 4.2 shows the run-time environment of a successful method invocation of a (single-threaded) Java program, containing a class Class Under Test (CUT) whose local history is specififed by an attribute grammar. The actors in the sequence diagrams are:

- 'User Prog': A client class that instantiates and uses CUT.
- 'Debugger': Java debugger that intercepts all method calls and corresponding returns from 'User Prog' to CUT.
- 'History (instance)': an instance of the history class. This class stores the local history of each object of CUT.
- 'Parser': an instance of a parser for the given attribute grammar. The source code of the Parser was generated by the Parser Generator.
- 'Assertion Checker': provides facilities to check assertions at run-time.
- 'Class Under Test (CUT)': The class which was specified using an attribute grammar.
- 'stderr': the standard error stream of the system. Error reports (such as an assertion failure or protocol violation) can be sent to this stream.

Figure 4.3 shows a scenario in which a method return causes the updated history to violate the grammar rules. In this case, the parser detects a parse error and outputs a protocol violation to 'stderr'. The scenario in which parsing is successfull, but the assertions cause an error, is not shown but very similar.

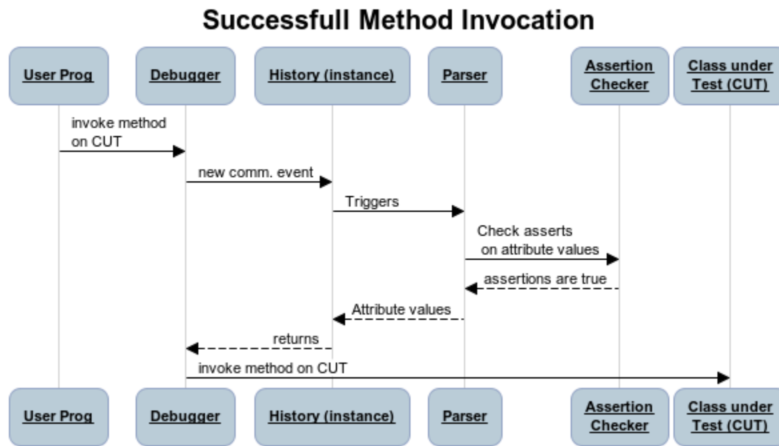

Figure 4.2: Run-time environment of successfull method invocation

4.1 Instantiating the 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 to what extent the requirements stated in the previous section are satisfied by these current tools. The main overhead of the run-time checker is caused by the parser, hence we discuss performance (both theoretical and in practice) in the paragraph on parser generators.

Meta-Programming Rascal [58] is a tool-supported domain specific language for meta programming. We use its parsing, source code analysis, source-to-source transformation and source code generation features. A ± 1000 line Rascal program[1] takes care of:

[1]Excluding the grammar for Java.

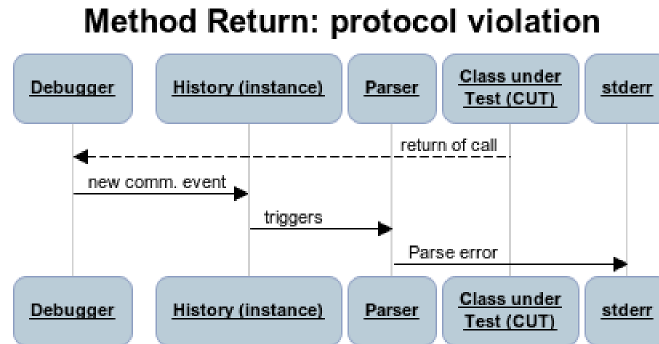

Figure 4.3: Run-time environment of successfull method invocation

- parsing and analyzing the Java method signatures in the communication view.
- generating Java source for a debugger. The debugger should intercept any method call and return, and inform the History class that an event occured.
- generating the token classes `call-m` and `return-m` for each call and return event in the view.
- generating the History class, which specifically accepts new events from the provided methods in the interface and acts as a token stream for the generated parser.

The full source code which Rascal generates for the above tasks contains about 50 times the number of events + 100 lines of code, in other words, the size of the generated code depends mainly on the number of events in the communication view.

Note that we require general meta programming features for several input languages, not just Java. This application of Rascal has three languages as input (ANTLR grammars, View declarations and Java), and one output language (Java). Rascal runs on a JVM, such that it integrates into any Java environment.

In the following Rascal snippet we generate update methods in the history class which are called whenever a method returns.

```
return "
<for (`<mods> <return> <id> (<formals>)` <- methods) {
  r = "return_<id>";>
public void update(return_<id> e) {
<if (r in tokens){>
  e.setType(<grammarName>Lexer.<tokens[r]>);
  addAndParse(e);<}>
}
<}>";
```

This return statement contains three levels. The Rascal language level (in boldface) provides the return statement, the string, and embedded in the string expressions marked by `<...>` angular brackets. The string that is generated represent an (unparsed) Java fragment. The fragments embedded in back ticks (`) represent parsed Java fragments from the input interface. Inside those fragments Rascal expressions occur again between angular brackets.

The string template language of Rascal allows us to instantiate a number of methods called `update` using a `for` loop and an `if` statement. The data that is used in the for loop is extracted directly from the parse trees of the methods in a Java interface file. The concrete Java source pattern between the back ticks (`) matches the declaration of a method in the interface, extracting the name of the method (`<id>`). Note that this snippet uses variables declared earlier, such as `tokens` which is a map from method names to token names taken from the view declaration in the interface and `grammarName` which was also extracted from the view earlier. Albeit complex code due to the many levels required for this task, the code is short and easy to adapt to other kinds of analysis and generation patterns.

The main disadvantages of Rascal are that it 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. The identified bugs were fixed quickly by the Rascal team, and its powerful parsing, pattern matching and transforming concrete syntax features proved indispensable.

Debugger 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 separate 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`. No actual Java source code for the class under test is needed for the debugging. The approach is safe in that no source code nor bytecode is modified for the monitoring. 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.

As an alternative we have also tested AspectJ, a Java compiler which supports aspect-oriented programming. Aspect-oriented programming is tailored for monitoring. AspectJ 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 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 can either be woven into Java source code, byte code or at class load-time fully automatically by AspectJ. Note that in contrast to the above Java Debugger approach this step involves changing the source or bytecode, which may be deemed as less safe. We use the inter-type declarations of AspectJ to store the local history of an object as a field in the

```
     /* 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));
 }
```

Figure 4.4: Aspect for the event 'call int read(char[] cbuf, int off, int len)'

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 4.4 shows a generated aspect. The second and third line specify the relevant method, in this case `BufferedReader.read`. The fourth line binds variables ('clr', 'cle', ...) to the appropriate objects. Note that to support dynamic binding, it is not possible to statically match method calls to in the Java source to the below pointcut: the dynamic type of the callee, which is determined at run-time, determines whether the pointcut matches. 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 garbage collected as soon as callee object itself is.

As a third alternative, we also tested the metaprogramming tool Rascal to generate code which intercepts the method calls and returns appropriately. This can be done by defining a transformation on the actual Java source code of the class under test, which 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 listed in Table 3.1. 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, dynamic binding, constructors and overloading. Moreover the meta-programming approach is also unsuitable if the Java source code is not available (which happens frequently for libraries) ing where only byte code is available, limiting the applicability of the tool. In summary, while it is possible to implement monitoring by defining a code transformation in Rascal, this rules out bytecode only libraries, and quickly gets complex due to the need for a full (up to date) Java grammar and the complexity of the full Java language.

Parser Generator For the the parser generator component we tested ANTLR v3, 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: productions which are only chosen during parsing whenever an associated Boolean expression (the condition) is true and allow for a degree of context-senstitiveness. Attribute grammars with conditional productions express protocols that depend on data which are typically not context-free. ANTLR also supports EBNF, a notation grammars which extends context-free grammars with the operations from regular expressions, for example the Kleene star. Though EBNF does not strictly increase expressiveness (the language generated by such grammars is still context-free), it is

convenient for practical purposes: sometimes a regular expression is simpler and more natural than a full-fledged grammar.

Due to the power of general context-free grammars extended with attributes (as introduced in the seminal paper [59] by Knuth), they can be quite expensive to parse. In particular, the currently best known algorithm [84] to parse context-free grammars has a time complexity of $\mathcal{O}(n^{2.38})$ (with very huge constants), where n is the number of terminals to parse. The current best practical algorithms (with reasonably sized constants) require cubic time. Clearly parsing n tokens cannot be done in less than $\mathcal{O}(n)$ steps, since the entire input must be read. Besides this trivial linear lower bound, no non-trivial lower bounds are known [41], though Lee [61] showed that multiplication of two square Boolean matrices can be reduced at a certain cost to parsing context-free grammars. In particular, she showed that if parsing n tokens can be done in $\mathcal{O}(n^{3-\epsilon})$ steps, then we can multiple two n by n Boolean matrices in $\mathcal{O}(n^{3-(\epsilon/3)})$ steps, with small constants. This means that any practical (i.e. small constants) sub-cubic parsing algorithm also can be used as a practical sub-cubic matrix multiplication algorithm. However no such fast practical algorithm is known for matrix multiplication.

ANTLR avoids the cubic-time parsing inefficiency by only supporting LL(*) grammars[2]. Due to the restriction, the parsing algorithm used by ANTLR is for most grammars linear, and quadratic in the worst case. A major disadvantage of ANTLR is that it lacks support for incremental parsing: each time the history is updated (i.e. a single terminal is added), the full history has to be reparsed. Additionally the full history has to be saved. Support for incremental parsing is planned by the ANTLR developers. We have not been able to find any Java parser generator which supports incremental parsing of attribute grammars.

Assertion Checker We tested two state-based assertion languages: standard Java assertions and the Java Modeling Language (JML). Both languages suffice for our purposes. A Java

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

assertions is a statement `assert b;` where `b` is a standard `boolean` expressions. As a consequence, note that Java assertions can contain calls to methods that return a `boolean`. Though Java assertions can not contain quantifiers, it is to some degree possible to simulate those using a method containing a loop. Java does not enforce assertions to be side-effect free: one needs to check manually that only 'pure' assertions are used.

JML is far more expressive than the standard Java assertions. It allows unbounded quantification, in general any first-order formula can be expressed in JML, and supports Design by Contract (see also Section 5.1). JML also ensures that assertions are side-effect free. Unfortunately the JML tool support is not ready yet 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 [19]. OpenJML, a new 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 of OpenJML at `http://sourceforge.net/tracker/?group_id=65346&atid=510629` for the kind of issues we have encountered when using OpenJML.