Prototyping a tool environment for run-time assertion checking in JML with Communication Histories

Frank S. de Boer
CWI
Science Park 123
Amsterdam, Netherlands
F.S.de.Boer@cwi.nl

Stijn de Gouw
CWI
Science Park 123
Amsterdam, Netherlands
C.P.T.de.Gouw@cwi.nl

Jurgen Vinju
CWI
Science Park 123
Amsterdam, Netherlands
Jurgen.Vinju@cwi.nl

ABSTRACT

In this paper we present prototype tool-support for the run-time assertion checking of the Java Modeling Language (JML) extended with communication histories specified by attribute grammars. Our tool suite integrates Rascal, a meta programming language and ANTLR, a popular parser generator. Rascal instantiates a generic model of history updates for a given Java program annotated with history specifications. ANTLR is used for the actual evaluation of history assertions.

Categories and Subject Descriptors

D2.1 [**SOFTWARE ENGINEERING**]: Requirements-/Specifications—*Tools*; D2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques; D2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification; D2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Tracing*; F3.1 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs; F4.2 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Grammars and Other Rewriting Systems

General Terms

Verification

Keywords

Interface Specification, Run-Time Assertion Checking, Attribute Grammars

1. INTRODUCTION

We present in this paper prototype tool-support for the run-time assertion checking of the Java Modeling Language (JML) [3] extended with communication histories specified by attribute grammars (as originated from [12]).

The main problem addressed in our paper is how to specify properties of communication sequences which form the very semantic foundations of object-oriented programming: In [10], a *fully abstract* trace semantics for a core Java-like language is given, where traces (or *communication histories*) are (finite) sequences of messages. A fully abstract semantics in general captures the externally observable behavior abstracting from irrelevant implementation details.

Communication sequences have also been used in proof systems for concurrent object-oriented languages: Dovland et al. [6] derived a complete proof system for concurrent objects from a sequential proof system using non-deterministic assignments over the local communication history. This work formed the basis for an extension of the KeY verification system for concurrent objects [1].

Using naively data structures like lists to represent such sequences however gives rise to very complicated assertions. In our paper, we represent these sequences as words of a language generated by a grammar. Grammars as such allow in a declarative and highly convenient manner to describe the protocol structure of the communication events.

However, the question now rises how to represent such grammars in JML assertions, and how to describe the data flow of the sequence. We argue that the data flow can be conveniently described by extending the grammar with attributes, providing a powerful separation of concerns between high-level protocol-oriented properties, which focus on the *kind* of messages sent and received, and data-oriented properties, which focus on the actual data communicated.

The main technical contribution of the paper is to show how the parser generated from the attribute grammar can be seamlessly integrated with JML assertions.

Our tool suite integrates Rascal, a meta programming language and ANTLR, a popular parser generator. Rascal instantiates a generic model of history updates for a given Java program annotated with history specifications. ANTLR is used for the actual evaluation of history assertions.

2. MODELING FRAMEWORK

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 invocation and completion of its methods (as declared by its interface).

In this section we describe our modeling framework in Java for the behavioral description of the interface of an object, expressed in terms of its communication histories. Our framework integrates into JML the use of attribute grammars for specifying properties of user-defined abstractions of communication histories. We explain the basic modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ECOOP '2010 Maribor, Slovenia EU

Copyright 2010 ACM 978-1-4503-0540-2/10/06 ...$10.00.

concepts in terms of the interface of a class implementing a stack as given in Figure 1:

```
interface Stack {
  void push(Object item);
  Object pop();
}
```

Figure 1: Stack Interface

Consider an instance of a class implementing this interface. The messages of the communication history of this object are modeled in our framework as instances of the following classes: 'call-push' and 'call-pop' which represent invocations of the methods 'push' and 'pop', and 'return-push' and 'return-pop' which represent the completion of the methods 'push' and 'pop'. Both classes 'call-push' and 'return-push' contain as an attribute the formal method parameter 'item' which stores the value of the actual parameter of the corresponding method invocation. The class 'return-pop' contains an attribute 'result' which stores the object returned. Our example demonstrates that it indeed is convenient to include the formal parameters as attributes also in messages which represent the completion of a method (see below).

A *communication view* is a general mechanism to introduce user-defined abstractions over the communication history in terms of the *terminals* of the (attribute) grammar used to specify its high-level (protocol) structure. For example, the communication view in Figure 2 introduces an abstraction of the communication history in terms of its *projection* onto the messages which correspond to the completion of the methods 'push' and 'pop'. We thus abstract in this particular case from the invocations of these methods and rename the selected messages 'PUSH' and 'POP', respectively.

The abstract behavior of this view can be defined in terms of sequences of the *terminals* 'PUSH' and 'POP' generated by the context-free grammar given in Figure 3.

This grammar describes the *prefix closure* of the standard grammar for *balanced* sequences of the terminals 'PUSH' and 'POP'. Note that in general the specification of the *ongoing* behavior of an object requires prefix closed grammars.

In order to specify the relation between the actual parameters of calls to the 'push' method and the result values of returns from the 'pop' method we introduce an *attribute* 'stack' of type EList, which extends the class List in Java with *side-effect free* implementations of the (overloaded) methods 'EList append(Object element)' and 'EList append(EList list)'. In Figure 4 we extend the grammar rules of Figure 3 with updates to the 'stack' attribute. Note that only the production rules for the non-terminal 's' are extended with updates to the attribute 'stack', since balanced stacks modeled by the non-terminal 'b' are empty. According to the formal semantics of attribute grammars, the value of the attribute 'stack' of a node in the parse tree labeled by the non-terminal 'S' is thus *synthesized* from the attributes of its descendant nodes. The built-in attributes of the leaf nodes of the parse tree are given by the message types of the corresponding terminals, as defined by the given communication view. As an example, 'item' is (in this case, the only) built-in attribute of the 'PUSH'-terminal. The attribute 'stack' of a sequence of the terminals 'PUSH' and

```
view StackHistory {
  return-push PUSH;
  return-pop POP;
}
```

Figure 2: Communication View of a Stack

s	::=	PUSH *s*
	|	*s s*
	|	b
b	::=	PUSH b POP
	|	ϵ

Figure 3: Abstract Stack Behavior

'POP' generated by the above grammar thus denotes the list of items pushed but not yet popped (i.e., the current content of the stack).

s	::=	PUSH S_1	stack = S_1.stack.append(PUSH.item)
	|	s_1 s_2	stack = s_1.stack.append(s_2.Stack)
	|	b	stack = stack.clear()

Figure 4: Attribute Grammar Stack Behavior

In order to use the attribute 'stack' of this grammar in assertions for specifying the contract of the 'push' and 'pop' methods of the 'Stack' interface (Figure 1) in terms of communication histories, the modeling framework provides a class 'StackHistory' which corresponds to the communication view of Figure 2. This class contains a 'getter' method 'EList stack()' corresponding to the attribute 'stack' defined by the grammar in Figure 4. In assertions specifying properties of the interface 'Stack' we refer to the communication history of the underlying implementation by a model field which denotes an instance of class 'StackHistory', as illustrated by the JML specification of the interface 'Stack' in Figure 5.

Here we assume that the class EList additionally provides side-effect free methods for returning the head and tail of a list, and comparing the contents of two lists. As described above, the identifier 'history' is declared in this interface specification as a *model field*. The JML keyword 'instance' indicates that each implementation of the interface 'Stack' will contain an instance variable 'history'. The above 'ensures' and 'requires' clauses describe the methods 'push' and 'pop' in terms of corresponding 'PUSH' and 'POP' operations on the stack abstraction of the communication history, specified by the above attribute grammar.

Figure 6 summarizes the main concepts of our framework for modeling communication histories by means of attribute grammars. The concept of 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. The attributes of the grammar are introduced in JML as attributes of a model field in the interface which represents the projection of the communication history of any implementation unto to the messages declared by the corresponding view. How this model field is *updated* in an implementation, is described in

```
interface Stack {
//@ public model instance StackHistory history;
//@ ensures history.stack().equals(
        \old(history.stack()).append(item));
void push(Object item);
//@ requires history.stack().size ! = 0;
//@ ensures history.stack().equals(
        \old(history.stack()).tail());
//@ ensures \result == \old(history.stack()).head();
Object pop()
```

Figure 5: JML Specification Stack Interface

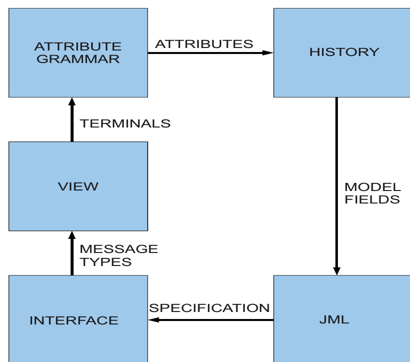

Figure 6: The Modeling Framework

the following section.

3. RUN-TIME ENVIRONMENT

We describe in this section the run-time environment of a single-threaded Java program annotated with JML-assertions over the communication history. The thread of control on a method invocation is depicted in the UML sequence diagram in Figure 7 and of a method completion in Figure 8.

The actors in both diagrams are:

- 'Program': contains an interface we wish to check, an implementation of this interface (StackImpl) and a client test class instantiating and using the implementation.

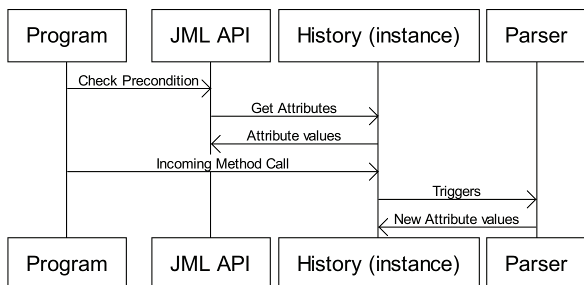

Figure 7: Sequence Diagram of method invocation

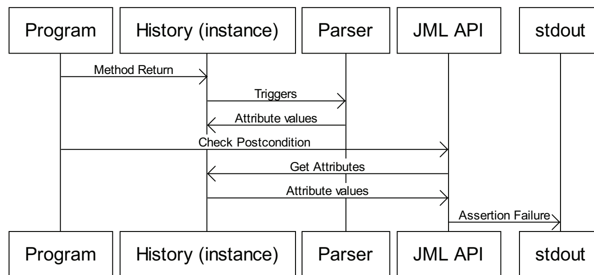

Figure 8: Sequence Diagram of postcondition assertion failure

- 'History (instance)': an instance of the history class. In the stack example this is an object of type 'StackHistory' storing the local history of a 'StackImpl' object.
- 'Parser': an instance of a parser for the given attribute grammar.
- 'JML API': provides facilities to check assertions at run-time.
- 'stdout': the standard output stream of the system. Error messages (such as an assertion failure) can be send to this stream.

Suppose a program invokes 'push' on a 'StackImpl' object s. We can distinguish between two communication events related to the 'push' method: an incoming method call to push, corresponding to the terminal 'PUSH' and a completion of the method call to push, which corresponds to the terminal 'POP'. We first describe the actions taken on the return of push and next explain how the situation differs for incoming method calls.

When 'push' returns (see Figure 8), a call to update the communication history of s is made since we capture returns of 'push' (according to the communication view in Figure 2). Based on the observation that the communication history can be considered to be a token stream (where the tokens are communication events), the object storing the local history of s triggers the parser to create the parse tree for this history, now viewed as a token stream. The parser also computes the values of the attributes of the non-terminals in the parse tree. Since parse errors correspond to violations of the protocol structure of the communication events imposed by the grammar, an assertion failure is generated on parse errors.

Once the parser returns it communicates to the history the values of the attributes of the root in the parse tree. Conceptually, these are the values of the history attributes describing properties of the data (actual parameters, return value) sent by the calls and returns that were recorded in the history. Finally, the method 'push' returns and a call to the JML API is done by the program to check the postcondition of 'push'. The JML API retrieves from '(History instance)' the values of the attributes of the communication history (in our case: 'history.stack()') used in the postcondition and checks the postcondition. If the postcondition is false, a message describing the Assertion Failure is send to 'stdout'.

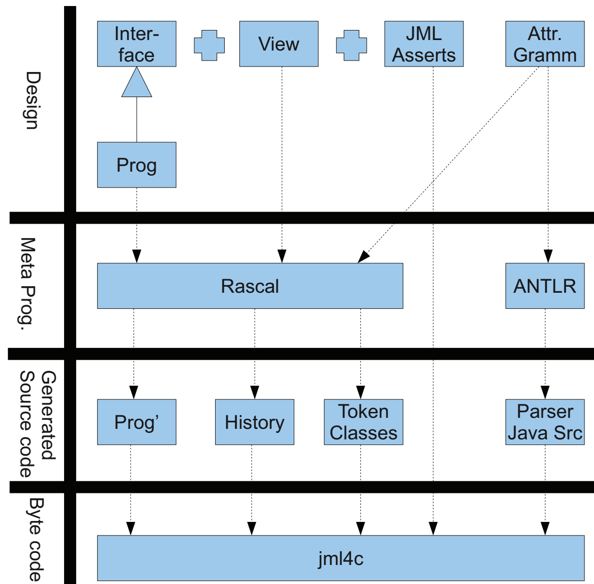

Figure 9: Generating the modeling framework

In contrast with returns where the assertion (the postcondition) is checked *after* updating the history, on method calls the assertion (the precondition) is checked *before* updating the history. This reflects the semantics of the JML requires-clause: only after the precondition is established, the actual incoming call is executed. A UML sequence diagram given in Figure 7 describes incoming method invocations.

A more in-depth discussion of the history class, the parser and the program transformations to add history updates to the program follows in the next section.

4. GENERATIVE FRAMEWORK

In this section we describe how the classes that implement our verification architecture are obtained from the given annotated source code and the model (shaped as an attribute grammar). We used Rascal [11] for general meta programming and ANTLR [14] for parser generation. See Figure 9 for an overview. We will describe the details of our experience with these tools in Section 4.3.

4.1 Manipulating models and source code

Rascal [11] is a domain specific language for meta programming. We use its parsing, source code analysis, source-to-source transformation and source code generation features. A ± 250 line Rascal program[1] takes care of:

- parsing and analyzing the source code of the Java/JML interfaces to extract the relevant method signatures and to extract the View declaration.
- parsing and transforming the source code of the Java implementations to weave in tracing calls at each entry and exit point of a method.
- generating the token classes for each call and return event for the methods in the Java interfaces.
- extracting the declared top-level attributes from the ANTLR grammar.
- generating the History class, which specifically accepts new events from the provided methods in the interface, allows retrieval of the specified attributes by the JML assertion checker, and acts as a token stream for the generated parser.

Note that we require general meta programming features for several input languages, not just aspect weaving for 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.

4.2 Generating parsers

The choice of a particular parser generator has consequences for the implementation of the History class, as well as for the kind of attribute grammars that can be used and the efficiency of the run-time checker. We have therefore identified several requirements guiding this choice.

- We need to be able to code attribute evaluation with a grammar.
- Since we interact with the annotated parse tree (in order to retrieve the attribute values of the root node), the *target language* for the parser should be *Java.*
- The terminals of the grammar are communication events, not string tokens. Thus the parser should be flexible enough to allow handling of any kind of stream of objects, or at least allow a correspondence between token types and other semantic values (the communication events).
- The type of a 'return-push'-token differs from a 'return-pop'-token. Therefore there must be support for heterogeneous token types: we must be able to specify a different token type for each abstract terminal in the grammar (or a different type for the semantic value of a token).
- Since we parse the communication history each time it is updated, parse trees for all prefixes are constructed. To improve efficiency, the parser should probably reuse parse trees of prefixes as much as possible, and avoid re-computation of attributes.
- If the user defines an attribute grammar for which no parser could be generated, no run-time checking can be done. Ideally the parser generator should be able to create a *parser for any context-free grammar.*

ANTLR facilitates many of these requirements. It generates fast recursive descent parsers for Java, allows encoding of attribute evaluation using grammar actions, and most importantly allows a custom stream of token classes.

Alas, it does not support incremental parsing or incremental attribute evaluation. As a result, our current prototype has two scalability issues. Firstly, the parsing of the history is done from scratch after each update of the trace. This means that while running a program, checking each assert

[1]Excluding the grammar for Java.

is in $O(h)$, where h is the size of the history. Secondly, the histories themselves will not easily fit into memory for industrially sized applications.

We postulate that our use of the attribute grammar formalism will help us approach these two scalability issues. Results on incremental parsing and attribute grammar evaluation, e.g. Hedin [7] and Hudson [8], should be applicable to allow us to evaluate asserts more independently of the size of the history. Instead of having to parse the entire history an incremental parsing and attribute evaluation mechanism computes the changes that have to be done to an attributed syntax tree after an edit action has been applied to the input string. Such an algorithm might therefore make our assertion checking more independent of the size of the history and solve our first scalability issue.

As an aside, since our attribute grammars are small and simple they produce simple abstract trees. It can be expected that many sub-trees are similar. Maximal sub-term sharing [16] (hash-consing) and dynamic programming techniques should help to exploit such redundancy. If a similar sub-tree is produced again and again, we may reuse the values of its attributes to skip redundant computations.

Related work on the use of context-free grammars for runtime verification also suggests that a scalable implementation is quite feasible [13]. This work extends and optimizes an LR parsing algorithm for checking context-free patterns in execution traces. The authors claim their prototype scales well on the DaCapo benchmark. Further study is needed to see if their approach can be extended to fit attribute evaluation.

4.3 Experience Report

Here we describe in more detail our use of ANTLR [14] and Rascal [11].

As explained above we need to pass custom token classes to the ANTLR generated parsers, which represent events in the communication history. This can be accomplished by sub-classing the ANTLR `Token` class. So, using Rascal we generate generate such sub-classes for every kind of event.

Values for the attributes of a non-terminal can be defined in ANTLR grammars using *semantic actions*: normal Java-code written inside brackets '{' and '}' which is attached to grammar rule. Inherited attributes can be passed as parameters of a non-terminal, synthesized attributes can be represented as return values of a non-terminal. There is no direct support for specifying heterogeneous token types, but one can cast a token in the grammar to the appropriate type when needed.

An important drawback of ANTLR is the lack of support for parsing arbitrary context-free grammars. ANTLR can only handle LL(*) grammars[2], which means that our stack grammar (in Figure 4) must be manually rewritten (factored) to accommodate the parser generator. For this purpose, additional attributes are used to count the number of surplus pushes ('$size') and we add the constraint that the number of pops may not exceed the number of preceding pushes at any point (facilitated by the *syntactic predicate* '{$s.ok}?', specifying '$s.ok' must be true):

The ANTLR adaptation of the Stack grammar is given in Figure 10. As ANTLR can not handle left-recursive gram-

[2]A strict subset of the context-free grammars. Left-recursive grammars are not LL(*). A precise definition can be found in [14].

```
grammar Stack;

//////////// start ::= s EOF ///////////
start returns [EList<Object> stack]
@init {
    $stack = new EList<Object>();
}
      : s[0, $stack] EOF {$s.ok}?
        { $stack = $s.completeStack; } ;
//////// s ::= PUSH s | POP s |  ////////
s         [int size, EList<Object> stack]
 returns [EList<Object> completeStack, boolean ok]
 : PUSH
   s1=s[$size+1, $stack.appendElement(
                     ((return_push)$PUSH).item())]
   { $completeStack = $s1.completeStack;
     $ok = ($s1.ok && $size>=0); }

 | POP
   s1=s[$size-1, $stack.tail()]
   { $completeStack = $s1.completeStack;
     $ok = ($s1.ok && $size>=0); }

 | { $completeStack = $stack;
     $ok = $size>=0; };
////////////////////////////////////////////////

PUSH: 'return_push';
POP:  'return_pop';
```

Figure 10: Stack grammar in ANTLR

mars, the grammar is right-recursive. With synthesized attributes alone, the definition of the 'stack' attribute in the production 's ::= POP s' is problematic, suggesting the use of an inherited attribute '$stack' to built up the stack, and a synthesized attribute '$completeStack' to propagate the full stack up the parse tree. The structural information over the 'PUSH', 'POP' events explicitly present in the original stack grammar is encoded in additional attributes ('$size' and '$ok') and is therefore highly obfuscated, also violating the separation of concerns that dictates the use of attributes to describe data-oriented (instead of protocol) properties. The grammar can be rewritten to an equivalent grammar without inherited attributes, but this further impairs readability.

This example clearly shows the importance of support for general CFG grammars. As ANTLR does not satisfy this requirement we are exploring other parsing generators now. There exist several general context-free parsing algorithms that could be applied. For example the SDF system [17] uses a General LR algorithm. There also exist incremental versions of GLR which may be applicable [18].

erience using the Rascal DSL shows that our application fits within its domain. The following code snippet illustrates this. We need information from Java interfaces and view declarations and the ANTLR grammar file, and use this to generate new Java code. In this 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 backticks (`) 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 backticks (`) 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.

5. CONCLUSION

We developed in this paper a modeling framework in Java for the integration of attribute grammars in JML and discussed the corresponding tool-support for run-time assertion checking. An open-source prototype can be downloaded from the author's website `http://www.cwi.nl/~cdegouw`.

Related Work.

Work on integrating assertions over communication history in Java programs has been the topic of Cheon and Perumandla in [5]. They present an extension of JML with call sequence assertions: regular expressions over method names, defining the set of allowed traces. A run-time checker is implemented to verify such assertions over the local history of an object. Data communicated by calls and returns is not considered. Hurlin [9] has elegantly extended this approach by using data (in particular, return values of calls) to parametrize the regular expression protocol specifications. Though no run-time checker is implemented (and no integration with JML), a static checker with proof rules for specifications in which the precondition has a specific form is provided.

Bartetzko et al. [2] introduce Jass, an alternative for JML with trace assertions. Trace assertions are given in a CSP-like notation and a precompiler translating trace assertions into Java 5 Annotations is provided. Trentelman and Huisman [15] describe a new formalism on top of JML for assertions using Temporal Logic formulae. 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 and develop a run-time assertion generator to accommodate run-time checking. In Chen et al. [4] also context-free grammars are used to recognize patterns in event histories. However a distinguishing feature of our approach is the use of attribute grammars to specify data-oriented properties of histories, and their full and seemless integration into JML. The work of Meredith et al. [13] is especially interesting from the optimization point of view (see Section 4).

Future Work.

We aim at the full development of tool-support for the run-time checking of assertions which refer to histories described by attribute grammars. As discussed in this paper, we aim at an integration of a parser generator which supports arbitrary context-free grammars and allows optimization of the run-time checking.

6. REFERENCES

[1] W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *Lecture Notes in Computer Science*, pages 387–406. Springer, 2009.

[2] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

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

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

[6] J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In D. Goldin and F. Arbab, editors, *Proc. of FInCo'07*, volume 203 of *ENTCS*, pages 19–34. Elsevier, May 2008.

[7] 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.

[8] S. E. Hudson. Incremental attribute evaluation: a flexible algorithm for lazy update. *ACM Trans. Program. Lang. Syst.*, 13:315–341, July 1991.

[9] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *SAC*, pages 587–592, 2009.

[10] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In S. Sagiv, editor, *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2005.

[11] 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.

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

[13] P. O. Meredith, D. Jin, F. Chen, and G. Rosu. Efficient monitoring of parametric context-free patterns. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 148–157, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[16] M. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

[17] M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002.

[18] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 31–43, New York, NY, USA, 1997. ACM.