Combining Monitoring with Run-Time Assertion Checking

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

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

Abstract. According to a study in 2002 commisioned by a US Department, software bugs annually costs the US economy an estimated $59 billion[1]. A more recent study in 2013 by Cambridge University estimated that the global cost has risen to $312 billion globally[2].

There exists various ways to prevent, isolate and fix software bugs, ranging from lightweight methods that are (semi)-automatic, to heavyweight methods that require significant user interaction. Our own method described in this tutorial is based on automated run-time checking of a combination of protocol- and data-oriented properties of object-oriented programs.

1 Run-Time Checking of Object-Oriented Programs

Given a program and a specification, a run-time verifier inserts checks in the code that determine whether the specification is satisfied. The checks are triggered during an actual execution of the program. In contrast to static verification, where properties are checked with respect to *all* executions (possibly there are infinitely many), run-time checkers only consider a single execution of the program. There is a wide range of specification languages used in run-time verification. They can be partitioned into two categories: languages that focus on the control-flow (these approaches are also called "monitoring"), and those focussing on data-flow.

As an example, one can use regular expressions to specify the order in which functions or methods in a program should be called [18]. Such specifications describe the control-flow of the program. Other formalisms for specifying control-flow are temporal logics, various kinds of automata and context-free grammars. For these formalisms, checking whether a given property holds of the current execution involves parsing a word (where the word is some representation of the trace of method calls in the current execution) in an automata. Generally only formalisms are chosen with a decidable parsing problem (in particular, this is the case for regular expressions, context-free grammars and most automata), so

[1] `http://web.archive.org/web/20090610052743/`
`http://www.nist.gov/public_affairs/releases/n02-10.htm`
[2] `http://www.prweb.com/releases/2013/1/prweb10298185.htm`

M. Bernardo et al. (Eds.): SFM 2014, LNCS 8483, pp. 217–262, 2014.

that everything can be automated. Specification languages for monitoring are discussed in more detail in the next section.

Approaches that specify data-flow usually do so by annotating the source code with assertions: logical formulas that must be true whenever control passes them. The formulas constrain the values of the program variables. If assertions are expressed in first-order logic with arithmetic, it is in general undecidable due to unbounded quantification (i.e. ranging over an infinite number of values) whether the assertion is true, thus usually the assertions are restricted in some way. For instance, Java contains an `assert`-statement which restricts to quantifier-free formulas (i.e. Boolean expressions). *Design by Contract* [53] provides a systematic way of using assertions to specify classes, interfaces and methods with respectively class invariants and pre- and postconditions. It was first used in the programming language Eiffel, and subsequently has also been applied to many other programming languages. For example, JML [14] is one of the most popular specification languages for Java and supports Design by Contract. JML also supports unbounded quantification, though assertions containing unbounded quantifiers are not checked by the JML run-time assertion checker.

While type checking for the most used imperative languages is done fully automatically at compile-time, run-time checking is done (also fully automatically) during execution, and properties are only checked for the current execution. This generally allows more expressive specifications compared to type checkers. Static verification cannot be automated. In particular, even if one restricts pre- and postconditions to just the formulas *true* and $false$, the resulting specification language is still undecidable (such assertions suffice to express the halting problem).

> Our own proposal is a method for run-time checking of object-oriented programs. We discuss below in more detail how run-time checking applies to the specific context of object-oriented programming, focussing first on single-threaded Java, and then describe an extension to concurrency.

Two of the basic features of object-oriented programming are data abstraction and encapsulation. In the design of software, these features support the methodology of *programming to interfaces* [31]. This methodology allows the developer of client code to abstract from irrelevant implementation details. Combined with the *design by contract* principle [53], programming by interfaces is one of the main approaches to mastering the complexity of software today.

One of the main formal behavioral interface specification languages for Java, the Java Modeling Language (JML) [14], is inherently *state-based*; i.e., JML mainly provides support for the specification of classes in terms of their fields, including so-called *model* fields that represent certain aspects of the data structures underlying the implementation. JML does not provide explicit support for the specification of the *interaction* between objects, in contrast to other formalisms such as message sequence charts and UML sequence diagrams [23,41].

On the other hand, the very semantic foundations of object-oriented programming are defined in terms of sequences of messages. In [43], 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 observable behavior abstracting from implementation details. Such an abstraction is required in for example a proper semantic definition of *behavioral subtyping* as is illustrated by the *fragile base class problem* [54]: According to the initial/final state semantics the class B (Figure 1) and its revised version in Figure 2 below are behaviorally equivalent.

```
class B {
  int x = 0;

  void m() {
    x = x+1;
  }

  void n() {
    x = x+1
  }
}
```

Fig. 1. First version of a base class B

```
class B {
  int x = 0;

  void m() {
    this.n();
  }

  void n() {
    x = x+1;
  }
}
```

Fig. 2. New version of a base class B

However the behavior of the subclass `M` defined in Figure 3 is clearly different for the two versions of the base class. In particular, when using the revised version of the base class, the definitions of the methods `m` and `n` in the subclass `M` are mutually recursive, giving rise to a non-terminating loop.

```
class M extends B {
  void n() {
    this.m();
  }
}
```

Fig. 3. Subclass of the base class

It is worthwhile to observe the analogy between this anomaly with repect to the substitutivity of (behaviorally) equivalent classes and the following basic counter-example to the compositionality of the initial/final state semantics for *multi-threaded* programs. Both threads T_1 and T_2 of Figure 4 have the same initial/final state semantics, however the initial/final state semantics of the *interleaving* of T_1 and thread T clearly differs from that of T_2 and T, if assignments are treated atomically.

```
thread T_1 { x=x+1; x=x+1 }
thread T_2 { x= x+2; }
thread T { x=0 }
```

Fig. 4. Multi-Threaded Programs

This counter-example shows that for a compositional semantics of multi-threaded programs we need more specific information about the underlying implementation, namely information about *how* the final state is generated from the initial state. The *minimal* information needed is captured by a fully abstract semantics (see [55] for a definition of the *full abstraction problem*). In general fully abstract semantics of concurrent systems are based on some form of *trace* semantics. Of interest here is that the above work on fully abstract semantics for a core Java-like language shows that some form of trace semantics is needed even for sequential (single threaded) programs. More specifically, [43] shows that a form of trace semantics for object-oriented programs indeed guarantees substitutivity assuming encapsulation of the object state. Consequently, also the fragile base class problem, as shown above, can only be resolved by some form of trace semantics of behavioral subtyping. In this case, the sequences of internal communication distinguishes the classes in Figure 1 and Figure 2. Fischer and Wehrheim [28] further investigate behavioral subtyping based on histories for object-oriented languages.

The following question arises: how to bridge the gap between the semantic foundations of Java based on traces and the abstraction level of formal behavioral interface specification state-based languages like JML? To this end we aim to find a formalism and corresponding tool support which:

1. Integrates properties of the control-flow and data-flow.
2. Is at the same abstraction level as the object-oriented programming model.
3. Is sufficiently expressive.
4. Is user-friendly, i.e., fairly close to the familiar surface syntax of the programming language.
5. Supports automated run-time checking.
6. Adds as little overhead as possible.
7. Contains some form of error reporting.

1.1 Outline

Section 2 contains a survey of existing formalisms and tools for specifying object-oriented programs.

Section 3 presents our own formalism for single-threaded object-oriented programs. The basic notions of a communication view, attribute grammars and assertions in attribute grammars are introduced. The section concludes with a motivation for the design choices that were taken during the development of the specification language.

Section 4 describes the architecture of SAGA, a tool for run-time checking the previously presented formalism. First, the components of a generic tool architecture are identified. Second, each component is instantiated with different tools which are then evaluated.

Section 5 contains two case studies. First we specify a small but very common Java library: a `Stack`. Subsequently we consider a larger industrial case from the e-commerce company Fredhopper. The section finishes with an evaluation based on the two cases.

2 Specifying Object-Oriented Programs: Formalisms and Tools

In this section we give an overview of existing specification languages for object-oriented programs. The specification languages can be roughly partitioned into those which focus on formalizing protocol-oriented properties (all but the last three categories listed below), and those focussing on data. All specification languages for protocol properties are based on some form of histories (also known as traces): sequences of method calls or returns. Languages focussing on data restrict the values of variables and fields in a program by means of logical formulas. We describe whether the specification languages are used in actual tools for static verification or run-time checking.

Sequence Diagrams. A sequence diagram[3] shows how multiple objects interact with each other over time. The diagram depicts the messages exchanged between

[3] See `http://www.omg.org/spec/UML/` for the latest UML specification of sequence diagrams.

the objects, and the order in which they are sent. In the context of object-oriented programs, the messages in a sequence diagram correspond to method calls. Since sequence diagrams visualize a single interaction, one could select a set of sequence diagrams as a specification of the behaviour of an object-oriented program, by requiring that the methods in the program are executed in the order specified by one of the sequence diagrams in the set. The resulting specification language describes properties of the protocol of the program.

While sequence diagrams have been used in theoretical studies for verification purposes [24,50], to the best of our knowledge, sequence diagrams as a specification language have not been used in actual tools for static or run-time verification. There are several reasons for this. First, any specification based on visualization tends to become unclear and even infeasible for describing large interactions. Second, the number of interactions exhibited in programs are often unbounded due to loops and recursion. Thus one would need an additional language for characterizing infinite sets of sequence diagrams.

Regular Expressions. A regular expression [44] is a declarative notation for a regular language. A language is a set of words. The words are usually (finite) strings of characters, though more complex objects can be used as well. The regular languages are those that can be obtained from a finite language by union, concatenation and Kleene star (an infinite union of finite concatenations of a language). If r_1 and r_2 are regular expressions, the notation for these three operations is respectively $r_1 + r_2$ (union), $r_1 r_2$ (concatenation) and r_1* (Kleene star). As an example, the regular expression $(ab)*$ denote the language of all words starting with “a” in which “a” and “b” alternate. The formal properties of regular languages have been widely studied in the field of formal languages and theory of computation, see for example the books [65,51].

As a specification language for object-oriented programs, regular expressions can be used to denote valid histories [18]. In this setting, the alphabet symbols correspond to method names, histories are represented as sequences of such alphabet symbols, and the valid histories are the words of the regular language. Note that in contrast to the previous sequence diagrams, regular expressions support a convenient notation for an infinite set of histories with the Kleene star.

There are various tools for run-time checking which support regular expressions: JmSeq [58], Tracematches [2] and JavaMOP [17]. The run-time check corresponds to solving the word problem (or parsing problem): decide whether the history is a word of the language denoted by a given regular expression. This can be done efficiently. In particular, if a history is valid according to a given regular expression, then parsing algorithms exist that decide in constant time whether the history resulting from appending a single call is also valid according to the regular expression (for the full history, this leads to parsing algorithms which are linear in the size of the history), see [32]. Moreover one does not need to store the full history, only the “state” of the parser for the previous history, and the method call which is added to the previous history are needed to determine validity of the new history.

Context-Free Grammars. A context-free grammar G is a quadruple $G = \langle V, \Sigma, P, S\rangle$ where V is a set of non-terminals, Σ is a set of terminal symbols, S is the start-symbol of the grammar (a non-terminal), and P is a set of production rules. The production rules specify how each non-terminal (independent of the context in which that non-terminal occurs, hence the name context-free) is allowed to be rewritten into a sequence of terminals and non-terminals. The grammar generates a context-free language, namely the set of all strings of terminal symbols that can be obtained by repeatedly applying the production rules of the grammar, starting from the start symbol of the grammar. For example, the grammar below (the used notation for the grammar is BNF [4]) with the non-terminal S as its start symbol, and "a" and "b" as terminal symbols generates all words of the form $\mathrm{a}^k\mathrm{b}^k$, $k \geq 0$ (in words: k a's, followed by k b's). The symbol ϵ denotes the empty word.

$$\boxed{\begin{array}{rl} S ::= & \mathrm{a}\ S\ \mathrm{b} \\ \mid & \epsilon \end{array}}$$

Context-free grammars are strictly more expressive than regular expressions. Using the so-called pumping lemma [65], one can prove that there is no regular expression which denotes the same language as the grammar above. However it is more complex to parse a string in a given context-free grammar, than in a regular expression. The currently best known practical algorithms can parse a string of length n in (worst case) $\mathcal{O}(n^3)$ time.

When used as a specification language for object-oriented programs, the terminal symbols are the method names, and the grammar specifies the valid orderings in which these methods are allowed to be called (in other words, the context-free grammar generates the valid histories). The run-time check which decides whether a history is valid consists of parsing the current history in the given grammar. PQL [52] and JavaMOP [17] are examples of tools that support run-time checking based on context-free grammars.

Automata. There are too many kinds of automata too list them here exhaustively, but all of them contain at least two things: a notion of a state, and a transition function between states. A finite automaton, one of the simplest automata, contains additionally a set of accepting states and a start state, with the requirement that the set of states must be finite. Finite automata are equivalent in expressive power to regular expressions. A push-down automaton is an extension of a finite automaton with a stack of infinite size. Push-down automata are equivalent in expressive power to context-free grammars.

In general, automata can be seen as a representation of a formal language: it takes a string as input, and accepts or rejects it based on an acceptance condition (the specific acceptance condition varies greatly between the different kinds of automata). However, unlike the above declarative formalisms of regular expressions and context-free grammars, automata tend to have an imperative flavor, focussing on *how* to parse a formal language, as opposed to directly specifying the language itself.

As a specification language for object-oriented programs, JavaMOP [17] supports finite automata. LARVA [22] supports a kind of automata called timed automata with stopwatches.

Temporal Logics. Temporal logic [60] is a variant of *Modal Logic* [30]. As the name indicates, the basis for temporal logics is a notion of time on which the truth of a formula may depend. In particular, as the system described a temporal logic formula evolves from one state to the next, the truth value of the formula *can* change. There are many kinds of temporal logics, but they can roughly be classified as being linear-time or branching-time. In linear-time logics, time is viewed as a set of paths (the paths being sequences of "time instances"). LTL [60] is a widely used linear-time logic. Branching-time logics represent time as a tree in which the current time is the root, and the branches are considered as "possible futures". CTL [20] is the main branching-time logic.

Temporal logics have been used extensively in model checking [21], for example in the tools (there are too many others to fully list here): BLAST [37] Java Pathfinder [70] NuSMV [19] PRISM [48] SPIN [40] UPPAAL [9]. Temporal logics have also been used in run-time checking, even for the functional language Haskell [66]. Examples of run-time checkers of temporal logic formulas for Java are JavaMOP [17] and Java Pathfinder [3].

Process Algebras. Process algebras [5,36] have been used to formally model concurrent systems. There exist a wide variety of process algebras (or process calculi), but all approaches share some basic characteristics.

Each approach has a notion of a basic process from which larger processes are built using various operators (for example, for parallel composition, sequential composition and recursion). Message passing is used as the only way two different actors or processes can interact (instead of for example, shared variable concurrency). Finally, all approaches come with a set of algebraic laws (hence the name "process algebra") which for example can be used to show that syntactically different processes are semantically equal (i.e. have the same behavior).

For reference we list some of the most used process algebras here: CSP [39,1], LOTOS [69], CCS [56], ACP [12] and the more recent π-calculus [57,64]. CSP has been used in the tool Jass [6] for run-time checking object-oriented programs.

First-Order Logic. First-order logic is a formal system for specifying and reasoning about formulas about objects (or values) that range over some domain of discourse. All variables and terms in a first-order formula range over objects of the domain of discourse.

First-order logic can be used to specify programs by means of assertions: a logical formula in which the free variables (i.e. all variables not bound by $\forall$ and $\exists$) are program variables. Assertions are written in the source code of the program and must be true whenever control passes over them. Floyd describes in [29] a method for proving properties using first-order assertions. His work was extended by Hoare in [38]. First-order logic also forms the basis for dynamic logic and second- and higher-order logic described below.

The popular tool-suite for JML [14] supports first-order assertions for both static verification and run-time checking of Java programs. The run-time checker for JML only checks formulas involving bounded quantifiers: quantified variables that range over a finite set of values. Validity of formulas involving unbounded quantifiers is in general undecidable, as already noted in the previous section.

Dynamic Logic. Like temporal logic, Dynamic Logic (DL) [62,33] is a variant of *modal logic* [30] which allows the direct expression of program equivalence and weakest preconditions. DL extends full first-order logic with two additional (mix-fix) relations: $< . > .$ (diamond) and $[.] .$ (box). In both cases, the first argument is a *statement*, whereas the second argument is another DL formula. A formula $< s > p$ is true if there exists a terminating execution of s after which the formula p is true. A formula $[s]p$ is true after all terminating executions of s, the formula p is true. For example, the formula $<$`x=x-1;`$>$ $(x == 0)$ is equivalent to $x = 1$. Dynamic logic has been used as a specification language in the static verifiers KeY [8] and KIV [35].

Second- and Higher-Order Logic. Second-Order logic is a highly expressive formalism which allows quantification over predicates and functions over the values of the underlying domain. This contrasts with first-order logic, in which only quantification over values of the domain is allowed. The expressiveness comes at a price: no sound and complete proof systems (with decidable proof rules and axioms) can exist for full second-order logic. Higher-Order logic is a generalization of second-order and first-order logic which allows quantification over objects of an arbitrary higher type (i.e. quantification over predicates of predicates, and so on). There exist various theorem provers for programs that support higher-order logic: Isabelle/HOL [45], Why3 [27], PVS [68] and Coq [13].

Another relatively recent approach is Separation Logic [63], which extensively uses inductively defined predicates (i.e. second-order logic), but adds several non-standard logical connectives to reason about heap properties, such as the separating conjunction and the points-to predicate. These connectives support modularity, though they complicate proof theory (they cannot be axiomatized [15]). Tools that support separation logic for static verification of programs include: VeriFAST [42], jStar [26], Slayer [11] and Smallfoot [10].

3 Trace Specifications for Control- and Data-Flow

The formalisms described in the previous section for specifying object-oriented programs can be categorized in roughly two categories: those focussing on the control-flow of the program, and those focussing on the data-flow of the program. Formalisms focussing on the control-flow specify the allowed orderings between method calls, for example using regular expressions, context-free grammars or temporal logics. Formalisms for describing the data-flow generally use assertions to restrict the values of fields, parameters or local variables, possibly enhanced by constructs such as pre-post conditions and class invariants for supporting design by contract. But none of described specification languages were developed to *combine* the specification of the control-flow with the data-flow in a single formalism. In contrast, the behavior of almost all Java programs depends on both control-flow and data-flow: for example, the behavior of a stack is fully characterized by the sequence of method calls to `push` and `pop` it receives (the control-flow), together with the parameter and return values (the data-flow). For Java programs that encapsulate their internal state[4] an execution can be represented by the *global communication history* of the program: the sequence of *messages* corresponding to the invocation and completion of (possibly static) methods, including actual parameters and return values. 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. Jeffrey and Rathke [43] develop a fully abstract semantics based on histories which coincides with the standard operational semantics.

Let us call the orderings between method-calls and returns the *control-flow* of a history, and the actual parameters and return values the *data-flow* of the history. In this section we develop a single formalism which allows combining data-oriented properties of the history with protocol-oriented properties. To be of practical use, such a formalism should be *user-friendly*, amenable to (at least) *automated run-time verification* and sufficiently *expressive*. Below we propose attribute grammars extended with assertions and conditional productions for the specification of histories, and compare several alternatives approaches with respect to expressiveness, usability and automation.

Specifications can be used in two different ways: as a description of how an API (in our case, a set of Java classes and interfaces) must be used by a client (this can be seen as a kind of formalized user manual), or as an internal specification for developers of a class to test the class which is being developed. In the first case, only methods visible to clients can be used in the specification (i.e. public methods and no self-calls, since the user has no control over private methods and self-calls), in the second case for internal use we must also monitor self-calls and calls to private methods.

3.1 Modeling Framework

The modeling framework consists of three basic ingredients: communication views, grammars with conditional productions, and assertions. We use the interface of the Java `BufferedReader` (Figure 5) as a running example to explain these modeling concepts. In particular, we formalize the following property of the `BufferedReader`:

[4] Encapsulation means that objects do not have direct access to the fields of other objects. If access to a field x is needed, the programmer instead adds two methods `T getx()` and `void setX(T val)`.

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

Note that the above property constrains the clients that use the BufferedReader; in other words, it is a kind of “user manual” for the reader, but does not guarantee that the reader itself works properly (since this property does not restrict the behavior of the reader itself). The property is a little unusual in that the reader actually cannot even detect whether a client uses it according to the above specification, since the reader has no way to detect whether the caller of close is the same object that constructed it. This last part can be seen as a form of dynamically checked ownership: the client which created the reader owns it, and the above property can serve as a first step to ensure that no information about the reader is leaked to other clients.

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

Fig. 5. Methods of the BufferedReader Interface

As a naive first step one might be tempted to define the behavior of BufferedReader objects simply in terms of ‘call-m($\overline{T}$)’ and ‘return-m($\overline{T}$)’ messages of all methods ‘m’ in its interface, where the parameter types $\overline{T}$ are included to distinguish between overloaded methods (such as read). However, interfaces in Java contain only signatures of provided methods: methods where the BufferedReader is the callee. Calls to these methods correspond to messages received by the object. In general the behavior of objects also depends on messages sent by that object (i.e. where the object is the caller), and on the particular constructor (with parameter values) that created the object. Moreover it is often useful to select a particular subset of method calls or returns, instead of using calls and returns to all methods (a partial or incomplete specification). Finally in referring to messages it is cumbersome to explicitly list the parameter types. A *communication view* addresses these issues.

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

Suppose we wish to formally specify the property on page 227. This is a property which must hold for the local history of all instances of `java.util.BufferedReader`. The communication view in Figure 6 selects the relevant messages and associates them with intuitive names: *open*, *read* and *close*.

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

Fig. 6. Communication view of a BufferedReader

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

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

If we would additionally include the above message in the communication view, all call-messages to the method `m` of class `C` sent by a `BufferedReader` would be selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Besides normal methods, communication views can contain signatures of constructors (i.e. the messages named *open* in our example view). As such, the set of signatures that occur in a communication view is not necessarily a subset of the signatures in the interface it specifies (since Java interfaces do not contain constructors). In this case, the view selects all calls/returns to an object of a class that implements that interface.

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

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

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

One may ask: why allow private methods to appear in specifications? After all, private methods cannot be used by an outside client of the class. The same question arises when considering whether to monitor self-calls or not. By allowing to monitor private methods and self-calls, the modeling framework and corresponding tool support can also be used by developers of the class, to test the current implementation of the class in development. Communication views include an optional excludeSelfCalls keyword which indicates per event whether self-calls must be tracked (for self-calls, the caller and the callee are the same). While typically developers do not want to exclude self-calls for the purpose of internal tests, this keyword is especially useful in public specifications for other clients, that describe how the class must be used by the client.

Local communication views, such as 6, selects messages sent and received by *a single object* of a particular class, indicated by 'specifies java.util.BufferedReader'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views. Figure 7 shows a global view which selects all returns of the method m of a class or interface (or any of its subclasses) called Ping, and all calls to m on a subtype of a class or interface called Pong. Note that communication views do not distinguish instances of the same class (e.g. calls to 'Ping' on two different objects of class 'Ping' both get mapped to the same terminal 'ping'). Different instances *can* be distinguished in the grammar using the built-in attributes 'caller' or 'callee', see the next two sections.

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

Fig. 7. Global communication view

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. See table 1 for a list of supported features which require special care. For example, to support dynamic binding, the actual run-time type of the callee must be used, instead of the static type of the variable or field in which the callee is stored. This means that the correspondence between the messages named in the communication view, and actual method calls in the program source code must be made at run-time. The other features listed in the table have been discussed above.

Table 1. Supported Java features that require special care

Constructors
Inheritance
Dynamic Binding
Overloading
Static Methods
Required Methods
Access Modifiers

Context-Free Grammars. Now that we have identified the basic messages using the communication view, the question arises how we can specify the valid orderings between these messages: *the protocol*. More specifically, we want to find a notation for the set of the valid histories (where a history is a finite sequence of messages). While the histories in this set will be finite (since at any point during execution, the then current history is finite), the set itself usually contains an infinite number of histories due to recursion or loops, so we cannot simply write it down explicitly. We can consider the set to be a language in which each history is a word, and each message is an alphabet symbol. This suggests we can use existing formalisms for defining languages, in particular the ones surveyed in Section 2. We use context-free grammars to specify the protocol behavior of histories.

Definition 1. *A history is valid with respect to a given context-free grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar.*

The discussion in Section 3.3 provides a motivation for choosing grammars over the other formalisms, and a justification for our definition of a valid history.

The grammar below specifies the valid histories of the `BufferedReader`:

$$\boxed{\begin{array}{rcl} S & ::= & \mathit{open}\ C \\ & \mid & \epsilon \\ C & ::= & \mathit{read}\ C \\ & \mid & \mathit{close}\ S \\ & \mid & \epsilon \end{array}}$$

Fig. 8. Context-Free Grammar which specifies that 'read' may only be called in between 'open' and 'close'

This grammar describes the prefix closure of sequences of the terminals 'open', 'read' and 'close' as given by the regular expression $((\mathit{open}\ \mathit{read}*\ \ \mathit{close})*)$. In general, the message names given by a communication view form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of messages (in particular, the start symbol S generates the valid histories).

3.2 Attribute Grammars and Assertions

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 first extend the above context-free grammars with so-called attributes.

Definition 2. *Terminal Attributes. Given a terminal T, an attribute of T assigns a value to each instance*[5] *of T (i.e. to each token of T).*

For example, consider a terminal INT_LITERAL, and suppose the string "33" is an instance of INT_LITERAL. One could define an attribute *val* for INT_LITERAL, which assigns the number 33 to the string "33". Note that terminal attributes can assign different values to different instances of the same terminal.

In the previous section we saw that (instances of) terminals correspond to call or return messages. The question arises: what are sensible attributes for such terminals? Several objects are involved in the sending of the messages: the

[5] A token is a string of symbols. A terminal can be seen as a token type, whose tokens are considered to be syntactically "similar".

caller, the *callee*, and the actual data being sent in the form of actual parameters or a return value *result*. We define *built-in* attributes (named *callee*, *caller*, and so on) to capture precisely those objects involved in the message. In summary, attributes of terminals are determined (i.e., built-in) from the method signatures given in the communication view.

Next we define attributes for non-terminals. Unlike attributes for terminals, they are defined by the user in the grammar. Given a context-free grammar G and a non-terminal V, let us denote by $L(V)$ the language generated from the non-terminal V by using the productions of G.

Definition 3. *Non-terminal Attributes. Given a set of values D and a context-free grammar with a non-terminal V, an attribute for V is a function* $f : L(V) \to D$.

Intuitively the above definition states that a non-terminal attribute assigns values to all of the words generated by that non-terminal. The value of non-terminal attributes is user-defined: the user must associate with each production, source code that computes the attribute values of all non-terminals involved in the production. There are two kinds of non-terminal attributes: synthesized attributes and inherited attributes. In each production the user defines the value of the synthesized attributes of the non-terminal on the left-hand side of the production, and the values of the inherited attributes of the non-terminals appearing on the right-hand side of the production. In general this does not rule out circular attribute definitions The seminal paper [47] in which Knuth first introduced attribute grammars contains an algorithm which detects circular definitions. Using actual source code for the attribute definitions ensures that all attribute values of non-terminals are computable. Of course this source code may not terminate, we rely on the user to make sure that it does.

In our setting, the grammar non-terminals generate sequences of call/return messages. Hence, a non-terminal attribute can be seen as a property of the data-flow of that sequence and hence, as an important special case, the attributes of the start symbol of the grammar can be considered as properties of the data-flow of the history. We are now ready to define attribute grammars:

Definition 4. *An attribute grammar is a pair* (G, F)*, where G is a context-free grammar, and F is a set of attributes for G.*

Note that 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 9 a user-defined synthesized attribute ‘c’ for the non-terminal ‘C’ is defined to store the identity of the object which closed the `BufferedReader` (and is `null` if the reader was not closed yet). Synthesized attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the ‘c’ attribute is not set in the productions of the start symbol ‘S’. The extension of context-free grammars to attribute grammars with assertions and conditional productions (next called "extended attribute grammars") naturally gives rise to the following modification in the definition of a valid history.

Definition 5. *A history is valid with respect to a given extended attribute-grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar, and all assertions in the grammar were true for every prefix of the history.*

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

```
S ::= open C1 {assert (open.caller == null ||
                       open.caller ==C1.c ||
                       C1.c == null);}
    | ϵ
C ::= read C1 (C.c =C1.c;)
    | close S (C.c = close.caller;)
    | ϵ       (C.c = null;)
```

Fig. 9. Attribute Grammar which specifies that 'read' may only be called in between 'open' and 'close', and the reader may only be closed by the object which opened it

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal 'call-m' can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears. Traditional pre- and post-conditions are still useful if in every context, the same assertion must be used: in that case, the assertions in the grammar would be duplicated at every occurence of the appropriate terminal. In Section 5.1 we show an example which uses traditional pre- and post-conditions.

It is important to note that for a meaningful semantics we have to restrict the attribute grammars to those grammars which are side-effect free (with respect to the heap) so that they don't affect the flow of control of the tested program, and which do not involve dereferencing of the built-in attributes of the grammar terminal (the formal parameters of the corresponding methods as specified by the communication view) because these refer to the *current* heap (and not to the past one corresponding to the occurrence of the message). This latter restriction is a fairly natural requirement as the method call which generated the grammar terminal only passed the the object identities of the actual parameters, but not the values of the fields of these objects. Note also that this requirement is automatically satisfied by using encapsulation.

Attribute grammars in combination with assertions cannot express *protocol that depend on data.* To express such protocols we consider attribute grammars enriched by *conditional productions* [59]. In such grammars, a production is chosen only when the given condition (a `boolean` expression over the inherited attributes) for that production is true. Hence conditions are evaluated before any of the symbols in the production are parsed, before synthesized attributes of the non-terminals appearing in the production are set and before assertions are evaluated. In contrast to assertions, conditions in productions affect the parsing process. The Worker grammar in Figure 30 in the case study contains a conditional production for the 'T' non-terminal.

In summary, a communication view selects and names the relevant messages. Selection allows to focus just on the relevant messages while names allow the identification of different messages, and enable the user to refer to the messages in a user-friendly manner. Context-free grammars specify the allowed orderings of the messages. The terminals of the grammars are the names as introduced by the communication view. These names are not just simple strings, but also contain various attributes such as the sender, receiver and the data sent in the message. The non-terminals are user-defined and generate sets of sequences of messages (i.e. histories), as given by the grammar productions. The start symbol of the grammar generates the valid histories. A context-free grammar can thus be seen as specifying a kind of invariant of the control-flow. Attribute grammars allow defining data properties of sequences of terminals, and in particular of the whole history. To this end, the user defines attributes of the grammar non-terminals in terms of the attributes of the grammar terminals. The values of non-terminal attributes are defined by Java code, which ensures that the attribute definitions are computable. The extension of attribute grammars with assertions makes it possible to specify data-oriented properties of the history, by constraining the value of the non-terminal attributes.

Finally, conditional productions can be used for protocols that *depend* on data. In general, it is possible to specify a single interface or class with multiple communication views (and corresponding grammars). This increases expressiveness: it makes it possible to specify the intersection of two context-free languages (if the user specifies two grammars, the history must satisfy both), and context-free languages are not closed under intersection. Furthermore multiple communication views and grammars can be used as partial specifications for the class or interface, to focussing on a particular behavioral aspect. If it is possible to decompose a single complete specification into multiple partial specifications, the resulting specifications are often simpler. This stems from the fact that a complete specification formalizes various properties, and care must be taken to avoid unwanted interference between these properties. In contrast, partial specifications can be used to formalize each property individually.

3.3 Discussion

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

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

Both the BufferedReader above and the case study use only regular grammars. Since regular grammars simplify parsing compared to context-free grammars, the question arises if we can reasonably restrict to regular grammars. Unfortunately this rules out many real-life use cases. For instance, the following grammar in EBNF[6] specifies the valid protocol behavior of a stack:

$$S ::= (\mathit{push}\ S\ \mathit{pop}\ ?)*$$

It is well-known that the language generated by the above grammar is not regular (apply the pumping lemma for regular languages [65]), so regular grammars (without attributes) cannot be used to enforce the safe use of a stack. It is possible to specify the stack using an attribute which counts the number of pushes and pops:

$$\boxed{\begin{array}{lll} S ::= & S_1\ \mathit{push} & (\texttt{S.cnt = S\_1.cnt+1;}) \\ \quad\ \ | & S_1\ \mathit{pop} & (\texttt{S.cnt = S\_1.cnt-1;}) \\ & & \{\texttt{assert S.cnt >=0;}\} \\ \quad\ \ | & \epsilon & (\texttt{S.cnt = 0;}) \end{array}}$$

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

[6] EBNF is an extension of the usual BNF notation for context-free grammars which allows using the operators on regular expressions (such as the Kleene star ‘*’ and the ‘?’ operator standing for an optional occurrence, i.e., ‘r?’ stands for ‘$r + \epsilon$’) directly inside grammars.

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

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

Proof. See [65].

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

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

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

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

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

$$\boxed{\begin{array}{rll} T ::= & S & \{\texttt{assert S.cnt >=0;}\} \\ S ::= & S_1\ \textit{push} & (\texttt{S.cnt = S\_1.cnt+1;}) \\ | & S_1\ \textit{pop} & (\texttt{S.cnt = S\_1.cnt-1;}) \\ | & \epsilon & (\texttt{S.cnt = 0;}) \end{array}}$$

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

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

Observe that our approach monitors only safety properties ('prevent bad behavior'), not liveness ('something good eventually happens'). This restriction is not specific to our approach: liveness properties in general cannot be rejected on any finite prefix of an execution, and monitoring only checks finite prefixes for violations of the specification. Most liveness properties fall in the class of the non-monitorable properties [61,7]. However it *is* possible to ensure liveness properties for terminating programs: they can then be reformulated as safety properties. For instance, suppose we want to guarantee that a method `void m()` is called before the program ends. Introduce the following global view

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

The occurence of the 'main' event (i.e. a return of the main method of the program) signifies the program is about to terminate. Define the EBNF grammar
$S ::= \epsilon$
$\quad | \quad m$
$\quad | \quad m+ \; main$
(where '+' stands for one or more repetitions). This grammar achieves the desired effect since the only terminating executions allowed are those containing `m`. In local views a similar effect is obtained by including the method `finalize` (which is called once the object will be detroyed) instead of `main`.

4 Implementation

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 first describe the generic architecture of our tool SAGA [25] 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

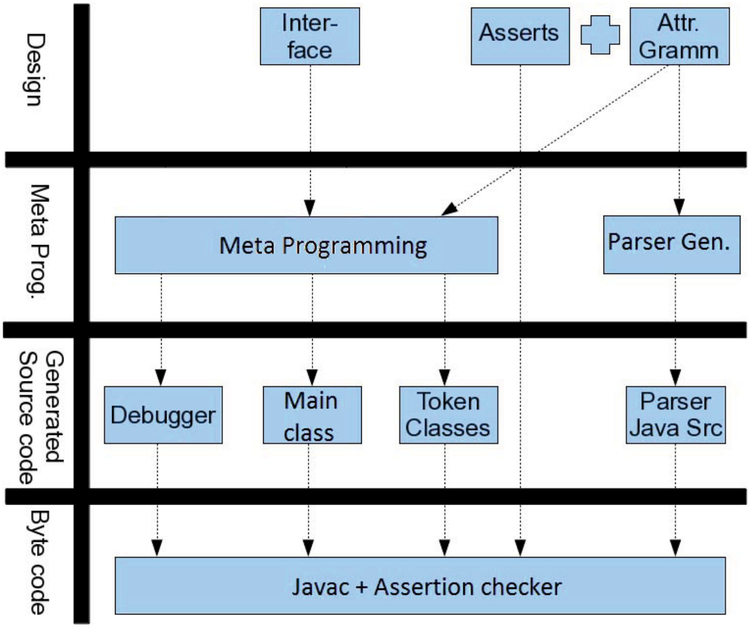

Fig. 10. Generic Tool Architecture

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.

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 [34]. 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.

To illustrate how the tools described above interact with each other at run-time, the UML sequence diagram in Figure 11 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 specifed 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.

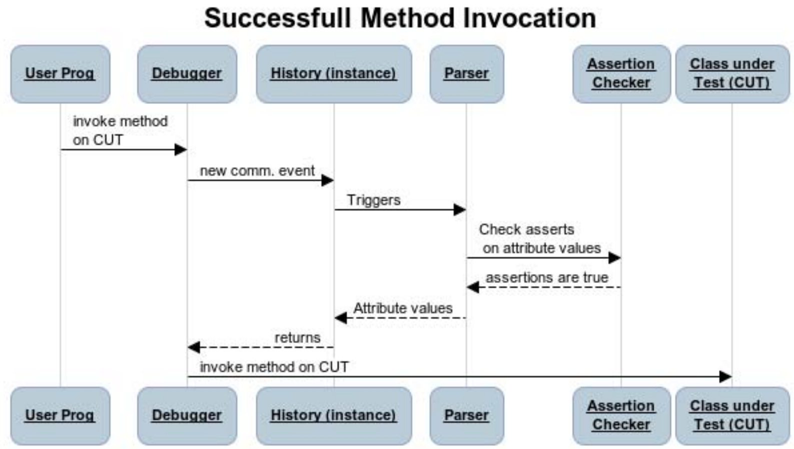

Fig. 11. Run-time environment of successfull method invocation

Figure 12 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.

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 [46] 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[8] takes care of:

- 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.

[8] Excluding the grammar for Java.

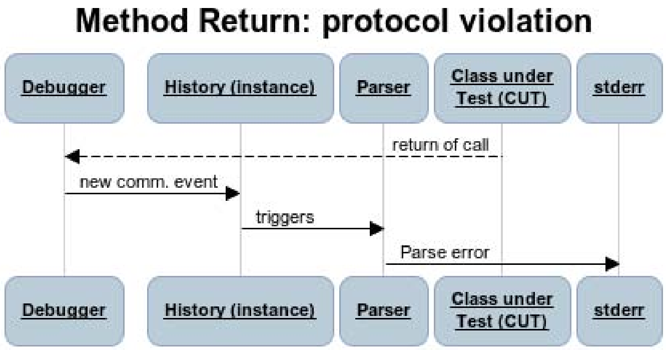

Fig. 12. Run-time environment of successfull method invocation

- 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 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 13 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.

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

Fig. 13. Aspect for the event ‘call int read(char[] cbuf, int off, int len)’

As a third alternative, we also tested the meta-programming 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 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 [47] by Knuth), they can be quite expensive to parse. In particular, the currently best known algorithm [67] 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 [32], though Lee [49] 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[9]. 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

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

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 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 [16]. 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.

5 Case Studies

In this section we use the formalism described in Section 3 and the extension to design by contract described in Section 4 to specify a Java library, and an industrial-sized case from the e-commerce company Fredhopper. The Java library we consider is a (last-in-first-out) Stack. The Stack example illustrates how the *Design by Contract* methodology as supported by JML can be used to specify the `push` and `pop`-methods purely in terms of histories in an elegant manner. In particular, this example shows how synthesized attributes of the start-symbol can be used conveniently inside method pre- and postconditions. Based on the case study, we discuss our experiences with SAGA.

5.1 Design by Contract: Stack

A Stack is an abstract data type which has only two operations `push` and `pop`. The operation `push` adds an object to the stack, while `pop` returns and removes

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

Fig. 14. Stack Interface

the last element from the stack which was pushed but not yet removed. The operation `pop` is not allowed on an empty Stack. Figure 14 shows an interface for the Stack in Java.

Our task is to find a specification for the Stack which ensures that `pop` is never called by the user on an empty stack, and moreover that `pop` returns the right object when called on a non-empty stack. The communication view in Figure 15 selects three events. The returns of push are needed to keep track of the elements which have been pushed onto the Stack. Note that it would incorrect be to consider the calls to `push` instead: suppose some strange implementation of `push` would itself call `pop` as its first action, before restoring the removed element and adding the element which was passed to `push`. Then calling `push` on an empty stack would fail (since that results in calling `pop` on an empty stack), but the history would be 'PUSH POP' (which seemingly looks valid for a Stack). Selecting returns of push avoids this problem. The calls to `pop`, which are referred to by the terminal 'POP', are needed to ensure that `pop` is never called on an empty Stack. In this case it would not suffice to track only returns of `pop`, since whenever `pop` is executed on an empty stack, the run-time checker would only detect the failure after executing of `pop` (which fails), and thus does not *prevent* unsafe behavior.

The protocol 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 16, where 's' is the start symbol.

The non-terminal 's' generates the *prefix closure* of the standard grammar for *balanced* sequences of 'PUSH' and 'POP' (which are generated by the non-terminal 'b'). This ensures that `pop` is never called on an empty stack.

In order to specify the relation between the actual parameters of calls to the `push` method and the return values of the `pop` method, we introduce a synthesized attribute 'stack' of type `JMLListValueNode` for the non-terminal 's'. `JMLListValueNode` is a JML class for a singly-linked list with *side-effect free* implementations of the methods `JMLListValueNode append(Object item)` , which appends an item to the list, and `JMLListValueNode concat(JMLListValueNode ls2)` which concatenates two lists. The intended value of the 'stack' attribute is a list of the elements which are pushed but have not yet been popped. Since balanced Stacks are empty, associating the 'stack' attribute also to the b-non-terminal would be redundant. Figure 17 shows how 'stack' is updated in each production of the non-terminal s. Intuitively the value of 'stack' at the root of the parse-tree

```
local view StackHistory specifies Stack {
    return push PUSH;
    call pop POP;
}
```

Fig. 15. Communication View of a Stack

$$
\begin{array}{lll}
\mathrm{s} & ::= & \mathrm{PUSH}\ s \\
 & \mid & s\ s \\
 & \mid & b \\
\mathrm{b} & ::= & \mathrm{PUSH}\ \ b\ \mathrm{POP} \\
 & \mid & \epsilon \\
 & \mid & b\ b
\end{array}
$$

Fig. 16. Abstract Stack Behavior

$$
\begin{array}{llll}
\mathrm{s} & ::= & \mathrm{PUSH}\ s_1 & (\texttt{stack =}s_1\texttt{.stack.append(PUSH.item);}) \\
 & \mid & s_1\ s_2 & (\texttt{stack =}s_1\texttt{.stack.append(}s_2\texttt{.Stack);}) \\
 & \mid & \mathrm{b} & (\texttt{stack = stack.clear();}) \\
\mathrm{b} & ::= & \mathrm{PUSH}\ \ b\ \mathrm{POP} & \\
 & \mid & \epsilon & \\
 & \mid & b\ b &
\end{array}
$$

Fig. 17. Attribute Grammar Stack Behavior

(i.e. an occurence of the start-symbol *s*) is a list containing the current contents of the Stack. Figure 18 shows the parse tree for the history resulting from the program `s.push(5); s.push(7); s.pop();`. Note that this does not mean that an actual implementation of the stack interface works correctly: the attribute grammar can be considered as a 'reference implementation' of the stack, but we still need to ensure that an actual implementation of the Stack matches (in the sense that calling `pop` returns the right value) this reference implementation.

In order to specify the method contracts for the Stack, the JML implementation of SAGA (described in Section 4.1) allows referring to the synthesized attributes of the root of the parse tree. Since the start symbol in the parse tree generates the whole history, intuitively the synthesized attributes of the start symbol can be thought of as a property of the entire history. In order to use the attribute 'stack' of this grammar in assertions for specifying the contracts of the `push` and `pop` methods of the 'Stack' interface (Figure 14) in terms of communication histories, the modeling framework provides a class `StackHistory` which corresponds to the communication view of Figure 15. This class contains a 'getter' method `JMLListValueNode stack()` which retrieves the value of the attribute 'stack' of the root of the parse tree of the current history.

Figure 19 illustrates how the `StackHistory` class can be used to specify the desired contracts. The JML keyword `model` indicates that `history` (of type `StackHistory`) can be used only in specifications. The keyword `instance`

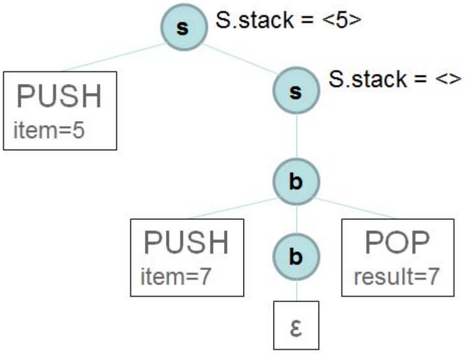

Fig. 18. Parse tree annotated with attribute values for the history push(5) push(7) pop() in the grammar of Figure 17 (irrelevant attributes ommitted)

```
interface Stack {
  //@ public model instance StackHistory history;

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).append(item));
  void push(Object item);

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).tail());
  //@ ensures \result == \old(history.stack()).head();
  Object pop();
}
```

Fig. 19. JML Specification Stack Interface

specifies that `history` will be added as a (non-static) field to any class that implements the `Stack` interface. The `ensures` and `requires` clauses specify the method contracts in terms of the 'stack' attribute (whose value is defined in the attribute grammar). Summarizingly, the property that `pop` may not be called on an empty stack is ensured by the productions of the grammar (the grammar productions can be considered to be an interface invariant for the protocol behavior), and the property that `pop` returns the right object is guaranteed by the method contracts and the definition of the attribute 'stack'.

Note that alternatively we could have avoided the method contracts by instead adding appropriate assertions in the attribute grammar before and after *every* occurence of 'PUSH' and 'POP' in the grammar. This leads to duplication since 'PUSH' occurs multiple times in the grammar. Moreover, for this alternative solution, we should also have added to the communication view that we intend to capture returns of `pop`: otherwise there would be no way to check that `pop` returned the right value. For the above example, we favour the above design-by-contract solution over the assertions-in-grammar, since it avoids duplication of specifications and additionally avoids adding the extra terminal for returns of pop. This increases readability of the grammar, and results in less overhead for the run-time check since the sequence of tokens to parse is shorter.

5.2 Fredhopper Case-Study

Fredhopper[10] is a search, merchandising and personalization solution provider, whose products are tailored to the needs of online businesses. Fredhopper operates behind the scenes of more than 100 of the largest online shops[11]. It provides the Fredhopper Access Server (FAS), which is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture (See Figure 20).

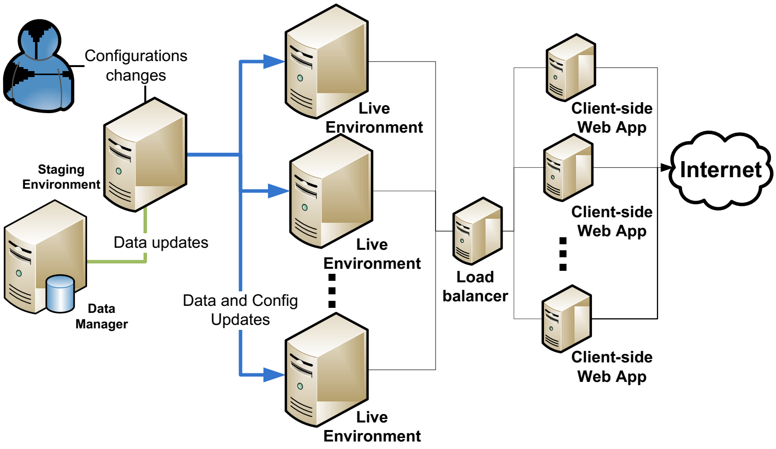

Fig. 20. An example FAS deployment

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

[10] http://www.sdl.com/products/fredhopper/

[11] http://www.sdl.com/campaign/wcm/gartner-maqic-quadrant-wcm-2013.html?campaignid=70160000000fSXu

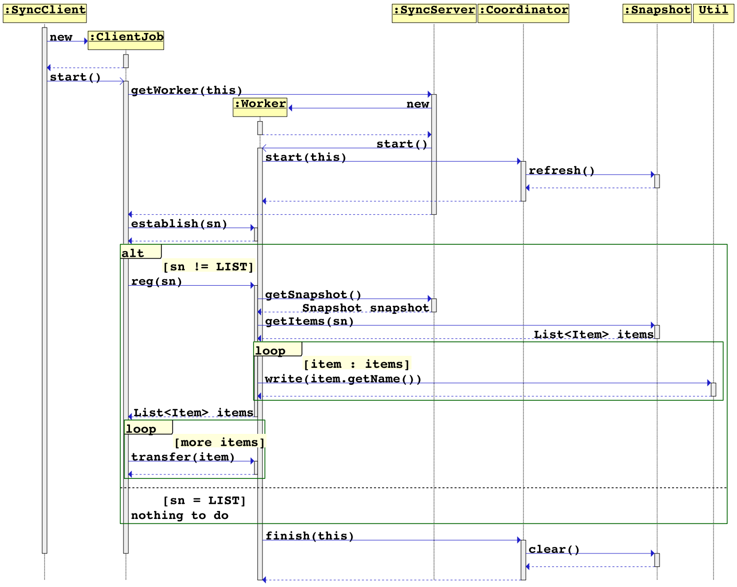

Fig. 21. Replication interaction

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 and write accesses of the staging environment's underlying file system, the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 21 shows a UML sequence diagram concerning parts of the replication protocol with the interaction between a SyncClient, a ClientJob, a Worker, a SyncServer, a Coordinator and a Snapshot. the diagram also shows a *Util* class that provides static methods for writing to and reading from `Stream`. The figure assumes that SyncClient has already established connection with a SyncServer and shows how a ClientJob from the SyncClient and a Worker from a SyncServer

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

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

Fig. 22. SnapShot and Worker interfaces of Replication System

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

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

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

Fig. 23. SyncServer and Coordinator interfaces of Replication System

are instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a *replication session*.

In this section we show how to modularly decompose object interaction behavior depicted by the UML sequence diagram in Figure 21 using SAGA. Figures 22 and 23 shows the corresponding interfaces and classes, note that we do not consider SyncClient as our interest is in object interactions of a replication session, that is after `ClientJob.start()` has been invoked.

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

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

Fig. 24. Snapshot Communication View

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

Fig. 25. Coordinator Communication View

- Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snapshot. Subsequently the invocations of methods `refresh` and `clear` should alternate.
- Coordinator: neither of methods `start` and `finish` may be invoked twice in a row with the same argument, and method `start` must be invoked before `finish` with the same argument can be invoked.
- Worker: `establish` must be called first. Furthermore `reg` may be called *if* the input argument of `establish` is not "LIST" but the name of a specific replication schedule, and that `reg` must take that name as an input argument. When the `reg` method is invoked and before the method returns, the Worker must obtain the replication items for that specific replication schedule via method `items` of the Snapshot object. The Snapshot object must be obtained via method `snapshot` of its SyncServer, which must be obtained via the method `server`. It must notify the name of each replication item to its interacting SyncClient. This notification behavior is implemented by the static method `write` of the class `Util`. The method `reg` also checks for the validity of each replication item and so the method must return a subset of the items provided by the method `items`. Finally `transfer` may be invoked after `reg`, one or more times, each time with a unique replication item, of type `Item`, from the list of replication items, of type `List<Item>`, returned from `reg`.

Figures 24 to 27 specifies communication views. They provide partial mappings from message types (method calls and returns) that are local to individual objects to grammar terminal symbols. Note that the specification of the Worker's behavior is modularly captured by two views: `WorkerHistory` and

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

Fig. 26. Worker Communication View

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

Fig. 27. WorkerReg Communication View

`WorkerRegHistory`. The view `WorkerHistory` exposes methods `establish`, `reg` and `transfer`. Using this view we would like to capture the overall valid interaction in which Worker is the callee of methods, and at the same time the view helps abstracting away the implementation detail of individual methods. The view `WorkerRegHistory`, on the other hand, captures the behavior inside `reg`. According to the informal description above, the view projects incoming method calls and returns of `reg`, outgoing method calls to `server` and `items`, and as well as the outgoing static method calls to `write`.

We now define the abstract behavior of the communication views, that is, the set of allowable sequences of interactions of objects restricted to those method calls and returns mapped in the views. Each local view also defines the file containing the attribute grammar, whoses terminal symbols the view maps method invocations and returns to. Specifically, Figures 28 to 31 shows the attribute grammars `Snapshot.g`, `Coordinator.g`,

$$\boxed{\begin{array}{l} S ::= \epsilon \mid rf\ T \\ T ::= \epsilon \mid cl\ S \end{array}}$$

Fig. 28. Snapshot Attribute Grammar

```
S ::=    T (T.ts = new HashSet();)
T ::= ϵ | st {assert ! T.ts.contains(st.t);}
          (T.ts.add(st.t);) T1 (T1.ts = T.ts;)
        | fn {assert T.ts.contains(fn.t);}
          (T.ts.remove(fn.t);) T1 (T1.ts = T.ts;)
```

Fig. 29. Coordinator Attribute Grammar

```
S ::= ϵ | et T (T.d = et.sn;)
T ::= ϵ | {!"LIST".equals(T.d);}?
          rg {assert rg.sn.equals(T.d);} U
U ::= ϵ | is V (V.m = new ArrayDeque(is.result);)
V ::= ϵ | tr   {assert V.m.peek().equals(tr.item);}
          (V.m.pop();) V1 (V1.m = V.m;)
```

Fig. 30. Worker Attribute Grammar

`Worker.g` and `WorkerReg.g` for views `SnapshotHistory`, `CoordinatorHistory`, `WorkerHistory` and `WorkerRegHistory` respectively.

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

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

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

- The input argument of `establish` and `reg` must be the same;
- `reg` can only be invoked if the input argument of `establish` is not "LIST";
- The return value of `reg` is a list of `Item` objects such that `transfer` is invoked with each of `Item` in that list from position 0 to the size of that list.

The grammar `WorkerReg.g` specifies the behavior of the method `reg` of Worker. It focuses on the invocations and returns of method `reg` of Worker as well as the outgoing method calls and returns of `Util.write` and `SyncServer.snapshot` and `Snapshot.items`. At the protocol level the grammar

```
/*S accepts call to Worker.reg() and, records */
/*the input schedule name, also S allows */
/*arbitary calls to SyncServer.snapshot() */
/*and Util.write() */
S  ::= $\epsilon$ | wr S | sp S | rg T (T.d = et.sn;)

/*T accepts and stores the return */
/*snapshot object from SyncServer.snapshot() */
T  ::= $\epsilon$ | sp V (V.d = T.d; U.s = sp.result;)

/*U ensures call items() is called on the same */
/*snapshot object, and the replication items */
/*for the correct schedule are retrieved */
U  ::= $\epsilon$ | ls {assert ls.callee.equals(U.s);
             assert ls.sn.equals(U.d);}
        V (V.s = U.s;)

/*V records replication items and their name */
/*returned from item() */
V  ::= $\epsilon$ | li W (W.is = new HashSet(li.result);
                W.ns = new HashSet();
                for (Item i :W.is) {
                  W.ns.add(i.name()); })

/*W ensures all replication */
/*items are processed */
W ::= $\epsilon$ | wr (W.ns.remove(wr.s);)
        $W_1$ ($W_1$.ns =W.ns; $W_1$.is =W.is;)
       | is {assert W.is.containsAll(is.result);
             assert W.ns.isEmpty();}
        X

X  ::= $\epsilon$ | sp X | rg X
```

Fig. 31. WorkerReg Attribute Grammar

specifies the regular expression (`snapshot items write`$*$) inside the invocation method `reg`. We use attribute definition to ensure the following:

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

– The returned list of `Item` objects from `reg` must be a subset of that returned from `Snapshot.items`.

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

5.3 Experiment

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

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

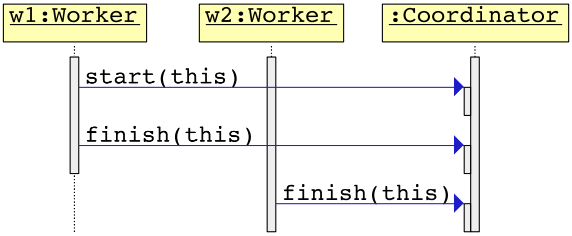

Fig. 32. Violating histories

To integrate of SAGA with the system tests, we employ Apache Maven tool[12], an open source Java based tool for managing dependencies between applications and for building dependency artifacts. Maven consists of a project object model (POM), a set of standards, a project lifecycle, and an extensible dependency

[12] `maven.apache.org`

```
class WKImpl extends Thread
implements Worker {
 final Coordinator c;
 WKImpl(Coordinator c) {
   this.c = c; }
 public void run() {
  try { .. c.start(this); ..
  } finally {
    c.finish(this); .. }}}
```

Fig. 33. Incorrect behavior of `WKImpl`

management and build system via plug-ins. We use its build system to automatically generate and package the parser/lexer of attribute grammars as well as aspects from views and grammars. We expose the packaged aspects, parser and lexer to FAS instance on the server farm and employ Aspectj using load-time weaver for monitoring method calls/returns during the execution of FAS instances on the server farm. Table 2 shows the number of join point matches during the execution of 766 replication sessions over live client data. Figure 34 shows the exection time of the 766 replication sessions with and without the integration of SAGA in milliseconds. At some points (for example, around 261 events), the figure seemingly indicates that the system runs faster with SAGA than without. In reality this is not the case: the dependence of the case study on user input (i.e., to start replication sessions) means that it is impossible to replicate an execution exactly (with the only difference being SAGA turned on and off respectively) and leads to small errors in the measurements. However, despite the fact that we cannot control the exact flow of control of the replication sessions (due to this dependence on user input), the graph clearly shows that the integration of SAGA has minimal performance impact on the execution time.

During this session we have found an assertion error at join point `call finish` due to the condition T`.ts.contains(`fn`.t)` not being satisfied at non-terminal T of the grammar Coordinator.g. Specifically, the implementation of Worker (`WKImpl`) that invoke `finish` before `start`. Figure 32 shows the sequence diagram of an invalid history causing the error, fully automatically generated from the output of SAGA. Figure 33 shows part of the implementation of `WKImpl`. It turns out that in the `run` method of `WKImpl`, the method `start` is invoked inside a `try` block while the method `finish` is invoked in the corresponding `finally` block. As a result when there is an exception being thrown by the execution preceding the invocation of `start` inside the `try` block, for example a network disruption, `finish` would be invoked without `start` being invoked.

Table 2. Join point matches in 766 replication sessions

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

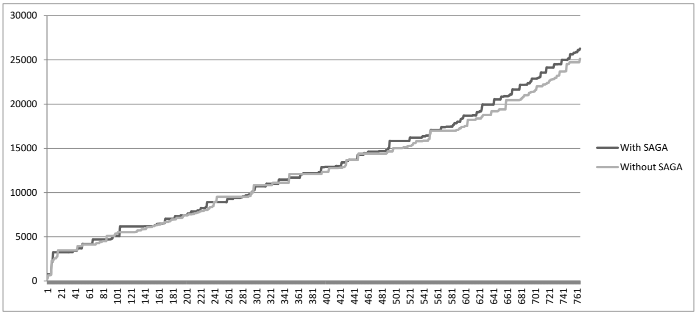

Fig. 34. Comparison of the execution time (milliseconds) of the replication sessions with and without the integration of SAGA

References

1. Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.): Communicating Sequential Processes. LNCS, vol. 3525. Springer, Heidelberg (2005)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA, pp. 345–364 (2005)
3. Artho, C., Drusinksy, D., Goldberg, A., Havelund, K., Lowry, M., Păsăreanu, C.S., Roşu, G., Visser, W.: Experiments with test case generation and runtime analysis. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 87–107. Springer, Heidelberg (2003)
4. Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In: IFIP Congress, pp. 125–131 (1959)

5. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes, 1st edn. Cambridge University Press, New York (2009)
6. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. Electr. Notes Theor. Comput. Sci. 55(2) (2001)
7. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
8. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
9. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a tool suite for automatic verification of real–time systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
10. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
11. Berdine, J., Cook, B., Ishtiaq, S.: sLAyer: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
12. Bergstra, J.A., Klop, J.W.: $\mathrm{Act}_{\mathrm{tau}}$: A universal axiom system for process specification. In: Algebraic Methods, pp. 447–463 (1987)
13. Bertot, Y., Castéran, P., Huet, G., Paulin-Mohring, C.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Texts in theoretical computer science. Springer, Berlin (2004)
14. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
15. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 108–119. Springer, Heidelberg (2001)
16. Chalin, P., James, P.R., Karabotsos, G.: JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 70–83. Springer, Heidelberg (2008)
17. Chen, F., Rosu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588 (2007)
18. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal 15(1), 7–25 (2007)
19. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
20. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
21. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
22. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM, pp. 33–37 (2009)
23. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)

24. de Boer, F.S., Bonsangue, M.M., Steffen, M., Ábrahám, E.: A fully abstract semantics for UML components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 49–69. Springer, Heidelberg (2005)
25. de Gouw, S., de Boer, F.S., Johnsen, E.B., Wong, P.Y.H.: Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study. In: SAC, pp. 1573–1578 (2013)
26. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA, pp. 213–226 (2008)
27. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
28. Fischer, C., Wehrheim, H.: Behavioural subtyping relations for Object-Oriented formalisms. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 469–483. Springer, Heidelberg (2000)
29. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, Providence, Rhode Island. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
30. Gabbay, D.M., Kurucz, A., Wolter, F., Zakharyaschev, M.: Many-Dimensional Modal Logics: Theory and Applications. Elsevier (2003)
31. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
32. Grune, D., Jacobs, C.J.: Parsing Techniques - A Practical Guide, 2nd edn. Springer (2008)
33. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
34. Hedin, G.: Incremental attribute evaluation with side-effects. In: Hammer, D. (ed.) CCHSC 1988. LNCS, vol. 371, pp. 175–189. Springer, Heidelberg (1989)
35. Heisel, M., Reif, W., Stephan, W.: Implementing verification strategies in the KIV-system. In: Lusk, E.‘., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 131–140. Springer, Heidelberg (1988)
36. Hennessy, M.: Algebraic theory of processes. MIT Press series in the foundations of computing. MIT Press (1988)
37. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
38. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
39. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
40. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
41. International Telecommunication Union. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). Technical report, ITU, Geneva (2001)
42. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
43. Jeffrey, A., Rathke, J.: ava Jr: Fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)
44. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata Studies (1956)

45. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Prog. Lang. Syst. 28(4), 619–695 (2006)
46. Klint, P., van der Storm, T., Vinju, J.J.: Rascal: a domain specific language for source code analysis and manipulation. In: Walenstein, A., Schupp, S. (eds.) Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177 (2009)
47. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)
48. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
49. Lee, L.: Fast context-free grammar parsing requires fast boolean matrix multiplication. J. ACM 49(1), 1–15 (2002)
50. Li, X., Liu, Z., He, J.: A formal semantics of UML sequence diagram. In: Australian Software Engineering Conference, pp. 168–177 (2004)
51. Martin, J.C.: Introduction to Languages and The Theory of Computation. McGraw-Hil (2010)
52. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a Program Query Language. In: OOPLSLA (2005)
53. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)
54. Mikhajlov, L., Sekerinski, E.: A study of the fragile base class problem. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 355–382. Springer, Heidelberg (1998)
55. Milner, R.: Fully abstract models of typed λ-calculi. Theoretical Comput. Sci. 4, 1–22 (1977)
56. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
57. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, New York (1999)
58. Nobakht, B., Bonsangue, M.M., de Boer, F.S., de Gouw, S.: Monitoring method call sequences using annotations. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 53–70. Springer, Heidelberg (2012)
59. Parr, T.J., Quong, R.W.: Adding semantic and syntactic predicates to LL(k): pred-LL(k). In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 263–277. Springer, Heidelberg (1994)
60. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
61. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
62. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: FOCS, pp. 109–121 (1976)
63. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
64. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
65. Sipser, M.: Introduction to the theory of computation. PWS Publishing Company (1997)
66. Stolz, V., Huch, F.: Runtime verification of concurrent Haskell programs. Electr. Notes Theor. Comput. Sci. 113, 201–216 (2005)
67. Valiant, L.G.: General context-free recognition in less than cubic time. J. Comput. Syst. Sci. 10(2), 308–315 (1975)

68. van den Berg, J., Jacobs, B.: The LOOP Compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
69. van Eijk, P.H.J., Vissers, C., Diaz, M. (eds.): Formal Description Technique Lotos: Results of the Esprit Sedos Project. Elsevier Science Inc., New York (1989)
70. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003)