J. of ASE, Volume 17(2), pp 149-180. 2010

Noname manuscript No.
(will be inserted by the editor)

Efficient Monitoring of Parametric Context-Free Patterns*

Patrick O'Neil Meredith · Dongyun Jin · Feng Chen · Grigore Roşu

the date of receipt and acceptance should be inserted later

Abstract Recent developments in runtime verification and monitoring show that parametric regular and temporal logic specifications can be efficiently monitored against large programs. However, these logics reduce to ordinary finite automata, limiting their expressivity. For example, neither can specify structured properties that refer to the call stack of the program. While context-free grammars (CFGs) are expressive and well-understood, existing techniques for monitoring CFGs generate large runtime overhead in real-life applications. This paper demonstrates that monitoring parametric CFGs is *practical* (with overhead on the order of 12% or lower in most cases). We present a monitor synthesis algorithm for CFGs based on an LR(1) parsing algorithm, modified to account for good prefix matching. In addition, a logic-independent mechanism is introduced to support matching against the suffixes of execution traces.

1 Introduction

Runtime verification (RV) is a relatively new formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against its requirements at runtime, as in testing. A large number of runtime verification approaches and systems, including TemporalRover (Drusinsky, 1997–2009), JPaX (Havelund and Roşu, 2001), JavaMaC (Kim et al, 2004), Hawk/Eagle

* This is an extended version of (Meredith et al, 2008), which has been supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

P. Meredith · D. Jin · F. Chen · G. Roşu
University of Illinois at Urbana-Champaign
E-mail: {pmeredit, djin3,-, grosu}@cs.uiuc.edu

(d'Amorim and Havelund, 2005), Tracematches (Allan et al, 2005; Avgustinov et al, 2007), J-Lo (Bodden, 2005), PQL (Martin et al, 2005), PTQL (Goldsmith et al, 2005), MOP (Chen and Roşu, 2007, 2003), Pal (Chaudhuri and Alur, 2007), RuleR (Barringer et al, 2008), etc., have been developed recently. In a runtime verification system, monitoring code is generated from the specified properties and integrated with the system one wishes to monitor. Therefore, a runtime verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a program instrumentor. The chosen specification formalism determines the expressivity of the runtime verification approach and/or system.

Monitoring safety properties is arbitrarily complex (Schneider, 2000). Recent developments in runtime verification, however, show that regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. As shown by a series of experiments in the context of Tracematches (Avgustinov et al, 2007) and JavaMOP (Chen and Roşu, 2007), parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 12% or lower. However, both regular expressions and temporal logics reduce to ordinary finite automata when monitored, so they have inherently limited expressivity. More specifically, most runtime verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such runtime verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. Examples of such structured safety properties include "a resource should be released in the same method which acquired it" or "a resource cannot be accessed if the unsafe method foo is in the current call stack".

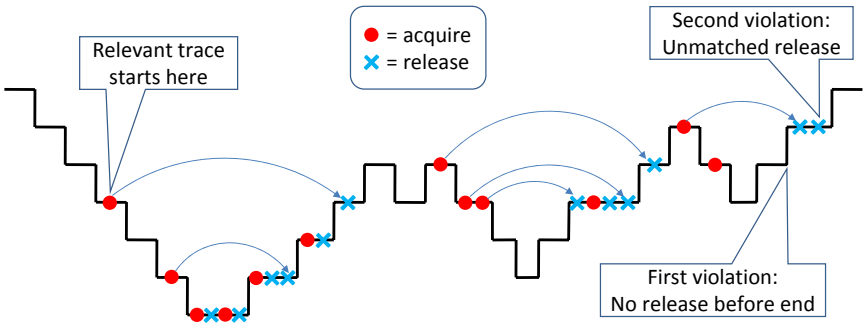

Fig. 1 Example trace for structured acquire and release of locks.

1.1 Example

An important and desirable category of properties that cannot be expressed using regular patterns is one in which pairs of events need to match each other, potentially in a nested way. For example, suppose that one prefers to use one's own locking mechanism for thread synchronization. As usual, for multiple reasons including the allowance of re-entrant synchronized methods (in particular to support recursion), locks are allowed to be acquired and released multiple times by any given thread. However, the lock is effectively released, so that other threads can acquire it, only when the lock releases match the lock acquires. One may want to impose an even stronger locking safety policy: the lock releases should match the lock acquires within the boundaries of each method call. This property is vacuously satisfied when locks are acquired and released in a structured manner using synchronized blocks or methods, like in Java 4+, but it may be easily violated when one implements one's own locking mechanism or uses the semaphores available in Java 5. For example, Figure 1 shows an execution violating this basic safety policy twice (each deeper level symbolizes a nested method invocation). First, the policy is violated when one returns from the last (nested) method invocation because one does not release the acquired lock. Second, the policy is also violated immediately after the return from the last method invocation because the lock is released twice by its caller, but acquired only once.

Supposing that the system is instrumented to emit events begin and end when methods of interest are started and terminated, and that the events acquire and release are triggered when the lock of interest is acquired and released, respectively, then here is an initial, straightforward way to express this safety policy as a context-free grammar:

$$S \rightarrow \epsilon \mid S \text{ begin } S \text{ end} \mid S \text{ acquire } S \text{ release}$$

Because of the production $S \rightarrow \epsilon$, the pattern is able to terminate (ϵ is the empty trace). This pattern will match any trace with begin events in balance with end events because these two events occur only in the production $S \rightarrow S$ begin S end[1], where they are matched. The S at the beginning of the production allows an unbounded number of these balanced groupings in a row, e.g., begin begin end end begin end is a valid trace with two balanced groupings in a row. The production $S \rightarrow S$ acquire S release is similar to that with begin and end events, allowing balanced groupings of acquire and release events. Because all the productions have the same recursive symbol, S, it is possible for begin/end pairs to nest within acquire/release pairs, and vice versa. Thus a valid trace would be begin acquire begin end release end begin acquire acquire release release end.

This pattern is simple and works, however, it has a deficiency in that it must monitor every begin of every method in a given program, even those which do not perform any thread synchronization. Next, we present a more

[1] Note that $S \rightarrow a \mid b$ is shorthand for $S \rightarrow a$, $S \rightarrow b$.

efficient grammar that ignores begin events that happen before the first acquire event in a trace [2]. The next grammar looks complicated, but we must stress that it is only complicated to improve monitoring efficiency.

$$\begin{aligned} S &\rightarrow \epsilon \mid S \text{ acquire } M \text{ release } A \\ M &\rightarrow \epsilon \mid M \text{ begin } M \text{ end} \mid M \text{ acquire } M \text{ release} \\ A &\rightarrow \epsilon \mid A \text{ begin} \mid A \text{ end} \end{aligned}$$

```
perthread SafeLock(Lock l) {
  event acquire before(Lock l) : call(* Lock.acquire()) && target(l) {}
  event release after(Lock l) : call(* Lock.release()) && target(l) {}
  event begin before() : execution(* *.*(..)) && !within(Lock) {}
  event end after() : execution(* *.*(..)) && !within(Lock) {}

  lr_lazy : S -> epsilon | S acquire M release A,
            M -> epsilon | M begin M end | M acquire M release,
            A -> epsilon | A begin | A end

  @fail { System.out.println("Unsafe lock operation found!"); } }
```

Fig. 2 JavaMOP specification for the safe lock safety property using the CFG plug-in.

Again, the productions begin with recursive references to S to allow for repetition of balanced groupings. This time, however, the S productions only allow for acquire and release, not begin and end. This ensures that only begins and ends occurring after the first acquire are monitored. The non-terminal M stands for "matched" sub-traces, i.e., traces in which all the pairs begin/end and acquire/release are properly matched, and A stands for sequences of (not necessarily matched) begin and end events. The A productions are necessary because failures would be reported for end events at the end of a trace due to the lack of a matching begin that occurred before the acquire creation event. The A productions also allow for more begin events after the last release because we do not want method calls after the last release to cause the invocation of the failure handler.

Any (finished or unfinished) execution trace that is not a prefix of a word in the language of S in the context-free grammar (CFG) above is an execution that violates the safety policy. The CFG runtime verification technique presented in this paper and implemented as a logic-plug-in in JavaMOP is able to monitor safety properties expressed as CFGs like above[3]. Monitoring-Oriented Programming (MOP) and JavaMOP (the Java implementation of MOP) are discussed in Section 3.

[2] Events that occur before the first event in a valid trace are ignored by the monitoring algorithm. Events which begin a valid trace are known as *creation events* (Chen and Roşu, 2007).

[3] Our CFG plug-in actually supports only the LR(1) and LALR(1) languages; when we use the term context-free we actually mean LR(1)/LALR(1), unless explicitly mentioned otherwise.

Figure 2 shows this SafeLock property expressed as a JavaMOP specification, using the CFG logic plug-in. The modifier perthread tells JavaMOP to consider events from separate threads as separate traces. This is particularly important as we do not wish begins and ends of separate threads to cause the pattern to fail. SafeLock denotes the name of the specification, while the list after SafeLock is the list of parameters to the specification (see below). SafeLock is parametric in the Lock because we do not wish the releases and acquires of separate Locks to interfere. The keyword event introduces an event; the event is first given a name, and then its trigger is defined using an AspectJ (Kiczales et al, 2001) advice (before the colon) and a pointcut (after the colon). Of particular note, however, is !within(Lock), used so that we do not monitor the begins and ends of the acquire and release methods, which would cause the pattern to fail. JavaMOP's generic approach to parametric specifications is described in Chen and Roşu (2009) and Chen et al (2009). Because of this generic approach, the logical formalisms in which properties are expressed need not be aware of the parameters; parameters are added automatically and generically by the JavaMOP framework.

The keyword lr_lazy introduces the CFG pattern. Three other possible keywords can be used : lr, lalr, and lalr_lazy. The two lazy keywords mean that when an event is encountered that causes a pattern match failure, the failure handler is invoked, but the event itself is not kept in the monitor state so that more failures can be found. If the error causing event were kept, as in the non-lazy keywords, each following event would cause an error, regardless of whether it should. The lazy method is how most programming language parsers work, allowing multiple syntax errors to be caught in one parse. lr and lalr determine which table generation algorithm is used (see Section 5 for more information on the two table generation algorithms). The first nonterminal in the pattern is assumed to be the start symbol of the grammar (see Section 5.1). Lastly, @fail introduces a *pattern failure handler*. The code within the braces following @fail runs whenever the pattern fails to match because an invalid event for a given point in a trace is seen. As an alternative, JavaMOP allows @match handlers. This gives extra power to our CFG plug-in because context-free languages are not closed under complementation.

The code generated automatically from the JavaMOP specification in Figure 2, following the technique described in the rest of the paper, has more than 700 lines of (human unreadable) AspectJ code. We ran this property against a hand-crafted program, which generated the sequence of events seen in Figure 1. Both pattern failures were successfully caught in a single run because the CFG plug-in does not add failure inducing events to the monitor state when lr_lazy is used. If the keyword lr is used instead of lr_lazy, only the first failure is caught by the generated monitor.

1.2 Contributions

Several approaches have been proposed to monitor context-free properties. For example, Program Query Language (PQL) (Martin et al, 2005) is based on a description language that encompasses the intersection of context-free languages. Hawk/Eagle (d'Amorim and Havelund, 2005) uses a fix-point logic and RuleR (Barringer et al, 2008) uses a rule based logic that can specify context-free properties. These approaches propose what we feel are rather complex solutions for monitoring parametric context-free patterns. They generate inefficient monitoring code in many cases, thus preventing practical parametric context-free property monitoring with these systems. The inefficiency of PQL in comparison to JavaMOP with context-free patterns is discussed in Section 6. This paper shows that monitoring (the LR(1) and LALR(1) subsets of) parametric context-free patterns is *practical*. We generate *non-parametric* monitors instead of parsers for the defined context-free pattern. Parameters are handled separately, using the algorithm in (Chen and Roşu, 2009; Chen et al, 2009). This way, we provide an efficient system for monitoring parametric context-free properties. Our algorithm is totally different from the monitoring algorithm used by the PQL system (Martin et al, 2005), which mixes the handling of parameters and monitoring of context-free patterns.

When monitoring pattern languages, such as extended regular expressions (ERE) or CFG, we wish to report a match anytime a trace at a given point in program execution matches the pattern. For example, if we have a pattern that is looking for writes to a closed file, we might use the ERE close write write*. We wish to report a match on every write, so that we can locate all of the trouble spots in the program. We call this matching every good prefix of the trace because close write is a prefix of close write write which is a prefix of close write write write, and we wish for a match to be reported on each of these prefix traces. We provide two methods to deal with the problem of monitoring good prefixes. One is to modify the LR(1) parsing algorithm with a *stack copying* process. The second method, called *guaranteed acceptance*, was discovered after our work in Meredith et al (2008).

Additionally, we extended JavaMOP with *suffix matching*. Suffix matching is, informally, matching against every suffix of a given trace, and is the mode of matching used in Tracematches (Avgustinov et al, 2007). A definition of suffix matching can be found in Section 4. We also describe optimizations particular to the JavaMOP suffix matching algorithm that improve the efficiency of suffix matching in JavaMOP.

We also performed an extensive evaluation of the CFG monitoring algorithm using the DaCapo (Blackburn et al, 2006) benchmark suite and properties used previously to evaluate runtime verification systems (Chen and Roşu, 2007; Bodden et al, 2007). The properties are expressed as CFGs in this evaluation rather than regular expressions for use in JavaMOP. Even when monitored using the CFG plug-in, however, these regular pattern based specifications still use constant space. We thus performed an evaluation of three strictly context-free properties – which use theoretically unbounded space – to

show that, even with such properties, the overhead is reasonable, and to show the usefulness of context-free properties. The results of this analysis compare favorably with PQL and Tracematches, two state-of-the-art runtime monitoring systems. One of these properties (ImprovedLeakingSync) is expressible in neither PQL nor Tracematches, for reasons explained in Section 6. Another of the properties (SafeFileWriter), while expressible in PQL, is not expressible in Tracematches because Tracematches has limited ability to express structured properties, rather than the full generality of the LR(1) languages.

Over both the adapted regular properties and the new strictly context-free properties, the overhead of JavaMOP with CFGs is, on average, over 8 times less than Tracematches on properties that Tracematches is able to express, and over 12 times less than PQL on properties that can be expressed in PQL. On all but 9 of the 45 benchmark/property pairs that generated events[4], the overhead is less than 5% in JavaMOP with CFGs.

Beyond the work of Meredith et al (2008), we have implemented three more versions of the original LR(1)-based cfg plug-in. We now support LALR(1) and a version of both LR(1) and LALR(1) that stay in an error state when an error token is encountered (using la/lalr instead of lr/lalr_lazy). We discovered and proved the concept of guaranteed acceptance (see Section 5.2.4). We also provide more a comprehensive analysis of the experiments as presented in Meredith et al (2008).

1.3 Paper Outline

The remainder of the paper is as follows: Section 2 illustrates related work; Section 3 gives a brief overview of MOP and JavaMOP; Section 4 describes suffix matching together with its novel, optimized implementation in JavaMOP; Section 5 explains our CFG monitor synthesis technique in JavaMOP, including considerations for suffix matching; Section 6 explains our experimental setup and the results of our experiments; Section 7 concludes the paper and describes some future work.

2 Related Work

2.1 Runtime Monitoring

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to Chen and Roşu (2007) for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP (Chen and Roşu, 2003; Chen et al, 2004; Chen and Roşu, 2007) have their specification formalisms hardwired and *only two of them* share the same logic (LTL). MOP will be

[4] Overall there are 66 benchmark/property pairs, but 21 of them generate no events, and are removed to more fairly represent the overhead of runtime monitoring.

Approach	Logic	Scope	Mode	Handler
JPaX (Havelund and Roşu, 2001)	LTL	class	offline	violation
TemporalRover (Drusinsky, 1997–2009)	MiTL	class	inline	violation
JavaMaC (Kim et al, 2004)	PastLTL	class	outline	violation
Hawk (d'Amorim and Havelund, 2005)	Eagle	global	inline	violation
RuleR (Barringer et al, 2008)	RuleR	global	inline	violation
Tracematches (Avgustinov et al, 2007)	Reg. Exp.	global	inline	validation
J-Lo (Bodden, 2005)	LTL	global	inline	violation
Pal (Chaudhuri and Alur, 2007)	modified Blast	global	inline	validation
PQL (Martin et al, 2005)	PQL	global	inline	validation
PTQL (Goldsmith et al, 2005)	SQL	global	outline	validation

Table 1 Runtime Verification Breakdown.

discussed in Section 3. This observation strengthens our belief underlying MOP — there probably is *no silver-bullet* specification formalism for all purposes. Also, most approaches focus on detecting either violations (pattern failures in CFG) or validations (pattern matches in CFG) of the desired property and support only fixed types of monitors, e.g., online monitors that run together with the monitored program or offline monitors that check the logged execution trace after program termination.

Specifically, there are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces)[5]. The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Table 1. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Of the systems mentioned in Table 1, only PQL (Martin et al, 2005), Hawk/Eagle (d'Amorim and Havelund, 2005), and RuleR (Barringer et al, 2008) can handle arbitrary context-free properties. Hawk/Eagle adopts a fixpoint logic and uses term rewriting during the monitoring, making it rather inefficient. It also has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not pub-

[5] Offline implies outline, and inline implies online.

licly available[6]. Because of this and the fact that Hawk/Eagle has not been run on DaCapo (Blackburn et al, 2006) with the same properties, we cannot compare our CFG plug-in with Hawk/Eagle. RulerR is a simplification of Eagle that is rule based rather than μ-calculus based, but it still has the ability to specify context-free properties. The current implementation is not built for efficiency or ease of expression with regards to context-free properties. In addition to PQL, we decided to perform comparisons with Tracematches (Avgustinov et al, 2007), as it is able to monitor a very limited set of context-free properties using compiler-specific support provided by their special AspectJ compiler, ABC (Avgustinov et al, 2005), and because it is a very efficient system. Pal (Chaudhuri and Alur, 2007) is able to monitor properties that take calls and returns into account, giving a limited context-free ability for this one case. Pal is implemented for C, rather than Java, and the implementation is not publicly available.

2.2 Context-free Grammars in Testing and Verification

Context-free grammars have seen use in several areas of testing and verification not immediately related to runtime monitoring.

Attributed context-free grammars were used as a means to generated test input and output pairs by Duncan and Hutchison (1981). The generated test inputs and outputs could be used both the test the specification from which the test grammar was designed, as well as the final implementation of a specification, using automatic test drivers. Their test case generator was capable of generating test cases from the grammar both randomly and systematically. The attributes of the context-free test generation grammars allow a user to attach context sensitive information to parts of the grammar, and allow for refinement of test case generation in order to avoid redundant test cases. Earlier attempts of test case generation via grammars (Purdom, 1972; Hanford, 1970; Houssais, 1977) were employed to generate test input only for compilers and parsers rather than programs and their specifications (though, Duncan (1978) used grammars to generate test cases in much the same way, it could not generate outputs, and it generated far too many similar test cases). In Maurer (1990) context-free grammars were used to generate test data for VLSI (very large scale integration) circuits. Sirer and Bershad (1999) applied the concept of test case generation using context-free grammars to Java virtual machine implementations.

All of these approaches differ quite a bit from runtime monitoring. The overhead of these approaches is not nearly so important because they are used to generate test cases in an offline manner, rather than running at the same

[6] (Avgustinov et al, 2007) makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public (d'Amorim and Havelund, 2005)), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from (Bodden et al, 2007). We have confirmed the inefficiency claims of (Avgustinov et al, 2007) with the authors of Hawk/Eagle.

time as a program that is under testing or a production system, situations for which runtime monitoring is intended. Additionally, runtime monitoring attempts to monitor behavior of a system rather than to generate test cases.

Hughes and Bultan (2007) used context-free grammars for an entirely different purpose that is more related to runtime monitoring than is test case generation. They created an interface specification language that uses context-free grammars to provide stub code for model checking. The grammar specifies the sequence of method invocations allowed by the component. The stubs are called by the code under model checking, providing a means of modular model checking. T The grammar generated stubs execute during the model checking process to ensure that the non-stub portions of code always follow the specification of method calls given by the grammar. In this way it is similar to runtime monitoring with context-free grammars, as the grammar is used to specify intended behavior, and flag errors when the behavior is not followed at runtime. Our work differs primarily in that it is designed to enforce behavior in a running system rather than to abstract a coponent, and in that it is parametric, whereas the interface grammars are not.

3 MOP Revisited

MOP is an extensible runtime verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an implementation of MOP for the Java programming language. By encapsulating our monitor synthesis algorithm for non-parametric CFG patterns in a JavaMOP logic plug-in, we have achieved an efficient monitoring tool for universally quantified parametric CFG specifications.

Additionally, we have implemented a novel extension of MOP, in JavaMOP, to support suffix matching independent of the language used for pattern specification. We define suffix matching as matching against every suffix of a given event trace, while total matching, also supported by JavaMOP, attempts to match the entire trace seen at a particular point[7]. Because Tracematches supports *only* suffix matching, and PQL supports a skip semantics more akin to suffix matching than total, we use suffix matching in our experiments.

3.1 MOP in a Nutshell

MOP (Chen and Roşu, 2003; Chen et al, 2004; Chen and Roşu, 2007) is a formal framework for software development and analysis, in which the developer specifies desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. Monitoring

[7] At a given point in program execution the trace of events seen at that point is evaluated as a complete trace in both total and suffix matching. This means if one is monitoring the pattern $\mathtt{e}^*$, a match must be reported every time the `e` event occurs.

code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system. MOP is a highly extensible and configurable runtime verification framework. The user is allowed to extend the MOP framework with his/her own logics via *logic plug-ins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture (Chen and Roşu, 2003), which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently.

In addition to choosing the formalism for a specification, one can also configure the behaviors of the generated monitor through different attributes (Chen et al, 2004). Depending upon configuration, the monitors can be separate programs reading events from a log file, from a socket, or from a buffer, or can be inlined within the program at the event observation points; monitors can verify the observed execution trace as a whole or check fragments of the trace. All these configurations are *independent* of the formalism used to specify the property. MOP also provides efficient and logic-independent support for universally quantified parameters (Chen et al, 2009), which is useful for specifying properties related to more than one object. This extension allows associating parameters with MOP specifications and generating efficient monitoring code from parametric specifications with monitor synthesis algorithms for non-parametric specifications. MOP's generic support for universally quantified patterns simplified our CFG plug-in's implementation.

The JavaMOP implementation provides several interfaces, including a web-based interface, a command-line interface, and an Eclipse-based GUI, providing the developer with different means to manage and process MOP specifications. JavaMOP follows a client-server architecture to flexibly support these various interfaces, as well as for portability reasons (Chen et al, 2006). AspectJ (Kiczales et al, 2001) is employed for monitor integration: JavaMOP translates outputs of logic-plug-ins into AspectJ code, which is then merged within the original program by the AspectJ compiler. Five logic-plug-ins are currently provided with JavaMOP: Java Modeling Language (JML) (Leavens et al, 2000), Extended Regular Expressions (ERE), Past-Time and Future-time Linear Temporal Logics (LTL) (see (Chen et al, 2006) for more details), and Context-Free Grammar (CFG) that is introduced in this paper. Note that these plug-ins can be supported by any implementation of MOP.

One might expect some loss of efficiency for MOP's genericity of logics. However, the JavaMOP-generated monitors yield very reasonable runtime overhead in practice, even for properties requiring intensive runtime checking (on the order of 12% or lower). In most cases it is as efficient as hand optimized monitoring code (Chen and Roşu, 2007).

4 Suffix Matching in JavaMOP

According to application requirements, one may want to check a property against either *the whole execution trace* or *every suffix of a trace*. Total matching has been adopted by many runtime verification approaches to detect pattern failures of properties, e.g., JPaX (Havelund and Roşu, 2001) and JavaMaC (Kim et al, 2004). Suffix matching has been used mainly by monitoring approaches that aim to find pattern matches of properties, e.g., Tracematches (Avgustinov et al, 2007). PQL has a skip semantics, wherein a specification is matched against the trace, but events may be skipped. A precise explanation of PQL's semantics is available in (Martin et al, 2005). To define suffix and total matching, we first must define traces and properties:

***Definition* 1** *Let $\mathcal{E}$ be a set of events. An* $\mathcal{E}$-trace*, or simply a* trace *when $\mathcal{E}$ is understood from context, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$.*

In the context of monitoring, an execution trace is a sequence of events observed up to the current moment, thus execution traces are always finite.

***Definition* 2** *An* $\mathcal{E}$-property *P, or simply a* property*, is a pair of disjoint sets (P_+, P_-) where $P_+ \subseteq \mathcal{E}^*$ and $P_- \subseteq \mathcal{E}^*$; P_+ is the set of* pattern matching traces *and P_- is its set of* pattern failing traces[8]*.*

Therefore, our notation of property is quite general; for each particular specification formalism, one needs to associate an appropriate property to each formula or pattern in that formalism. For example, for a CFG G, we let $P_G = (P_+, P_-)$ be defined as expected: P_+ is $L(G)$ (the language of G, see Section 5.1) and $w \in P_-$ iff w is not the prefix of any $w' \in L(G)$

***Definition* 3** *The* total matching semantics *of P is a function*

$$[\![P]\!]_{total} : \mathcal{E}^* \to \{match, fail, ?\}$$

defined as follows for each $w \in \mathcal{E}^*$:

$$[\![P]\!]_{total}(w) = \begin{cases} match & \text{if } w \in P_+ \\ fail & \text{if } w \in P_- \\ ? & \text{otherwise} \end{cases}$$

The suffix matching semantics *of P is a function*

$$[\![P]\!]_{suffix} : \mathcal{E}^* \to \{match, ?\}$$

defined as follows for each $w \in \mathcal{E}^*$:

$$[\![P]\!]_{suffix}(w) = \begin{cases} match & \text{if there are } w_1, w_2 \text{ such that } w = w_1 w_2 \text{ and} \\ & \quad [\![P]\!]_{total}(w_2) = match \\ ? & \text{otherwise} \end{cases}$$

[8] More recently we have generalized this concept to generic categories beyond just match and fail (Chen and Roşu, 2009; Chen et al, 2009). However, match and fail are sufficient for context-free patterns.

As an example of where suffix matching is useful, consider the HasNext property. This property specifies that the Java API method hasNext must be called before every call of next for an Iterator. If we use total matching and a match handler, we must define the pattern as (using a regular expression) "(hasNext + (hasNext next))* next next", to allow for all of the hasNext events, or correct uses of next that may occur before the two temporally adjacent calls to next. If we use suffix matching, because all suffixes of the trace are tried, the pattern "next next" is sufficient, so long as the hasNext event is still defined.

The previous design of JavaMOP supported only total matching. We have implemented a logic-independent extension of JavaMOP to also support suffix matching. This extension is based on the observation that, although total matching and suffix matching have inherently different semantics, it is not difficult to support suffix matching in a total matching setting, if one maintains *a set of monitor states* during monitoring and *creates a new monitor instance at each event* (this amounts to checking the property on each suffix incrementally). However, the situation becomes more complicated when one wants to develop a logic-independent solution, since different logical formalisms can have different state representations. For example, the monitor state can be an integer when the monitor is based on a state machine, a vector like the past-time LTL monitor, or a stack such as the CFG monitor discussed below. Hence, our solution is to *treat every monitor as a blackbox* without assumptions on its internal state. Also, instead of maintaining a set of monitor states in the monitor, we use a wrapper monitor that keeps a set of total matching monitors as its state for suffix matching. For simplicity, from now on, when we say "monitor" without specific constraints, we mean the monitor generated for total matching. When an event is received, the wrapper monitor for suffix matching operates as follows:

1. create a new monitor and add it to the "suffix matching" monitor set;
2. invoke every monitor in the monitor set to handle the received event;
3. if a monitor enters its "pattern fail" state, remove it from the monitor set;
4. if a monitor enters its "pattern match" state, report the pattern match.

The third step is used to keep the "suffix matching" monitor set small by removing unnecessary monitors. Indeed, this implements suffix matching semantics because each total monitor is monitoring a suffix of the current trace, and "pattern match" is only reported if one of the suffixes is valid.

Using our current implementation of suffix matching in JavaMOP, one may further improve the monitoring efficiency if the monitor provides an equals method that compares two monitors with regard to their internal states, and a hashCode method used to reduce the amount of calls to equals. This interface is used to populate a Java HashSet: the combination of the definition of hashCode and equals ensures the monitors in the HashSet are declared duplicates, and removed, based on monitor state rather than memory location. This interface can be easily generated by each JavaMOP logic plug-in because it has full knowledge of the monitor semantics. It is important to note that our approach does *not depend on the underlying specification formalism.*

JavaMOP requires that the logic plug-in designate *creation events* that are the starting events of a validating trace, in order avoid creating unnecessary monitors. A new monitor instance needs to be created only when creation events occur. This feature is especially useful when combined with suffix matching, which otherwise requires creating a new monitor at every event.

5 Context-Free Patterns in JavaMOP

We support the LR(1) subset of context-free grammars (CFGs), as well as LALR(1) which is a subset of LR(1). LR(1) is so named because it parses input **L**eft to right and produces a **R**ight-most derivation. The 1 denotes that one token of look-ahead is used. The LA in LALR stands for look-ahead, because, under certain conditions,states in the LR(1) table with different look-aheads may be merged in the LALR(1) table (see Section 5.2.3).

LR(1) can only recognize a subset of the deterministic context-free languages, which are themselves a strict subset of the context-free languages (CFLs) (Aho et al, 1986; Hopcroft et al, 2001). LR(1), however, is an expressive subset, able to define the syntaxes of most modern programming languages. We chose LR rather than LL because LR recognizes a larger number of grammars without translation. We base our implementation on the Knuth algorithm (Knuth, 1965) for LR(1) parser table generation as presented in (Aho et al, 1986). While the "action" and "goto" tables generated are normal LR(1) "action" and "goto" tables, the algorithm used to parse has been modified to work in the context of monitoring, explained in detail below. We added a plug-in, which generates LALR(1) using the algorithm presented in (Aho et al, 1986), because LALR(1) generates smaller tables in some cases. The LALR(1) tables are, at worst, identical to the LR(1) tables; they are never larger. The downside of LALR(1), however, is that it is a strict subset of LR(1). Comparisons between the table size of LR(1) and LALR(1) for the properties we tested can be found in Section 6, and an explanation of the LALR(1) optimization can be found in Section 5.2.3. Sections 5.1–5.2.3 cover standard issues related to LR parsing from a monitoring context, while the remainder of Section 5 covers new issues specific to adapting LR parsing to monitoring.

5.1 Preliminaries

A CFG G is defined as a tuple of the form, $G = (NT, \Sigma, P, S)$. Σ, the alphabet of the CFG, is often referred to as the set of terminals. A very special terminal, \$, represents the *end* of the input. NT is the set of nonterminals. P is the set of productions, which define what strings nonterminals derive. $NT \cup \Sigma$ is often called the set of symbols. Productions have the form $A \rightarrow \gamma$, where $A \in NT$ and γ is a string that either consists of symbols, or is the empty string, ϵ, i.e. $\gamma \in (\Sigma \cup NT)^*$. We use the conventional alternation operator, "|": a production

of the form $A \rightarrow \gamma_0|\gamma_1$ can be equivalently represented as two productions $A \rightarrow \gamma_0$ and $A \rightarrow \gamma_1$. S is the start symbol – that non-terminal from which all strings in the language are derived. For example, $G = (\{A\}, \{a, b\}, P_0, A)$ where $P_0 = \{A \rightarrow aAb|\epsilon\}$ is a simple CFG for the language $\{a^n b^n | n \in \mathbb{N}\}$. The non-terminal A can derive aAb an indeterminate amount of times before deriving ϵ, allowing $a^n b^n$ for any $n \in \mathbb{N}$.

Two important sets are defined for every non-terminal in a grammar: the *first* and *follow* sets. These are used in "action" table construction. The *first* set will be used to decide which terminals in the given grammar define monitor creation events (we shall be more specific about this below). The *follow* set will be useful in illustrating the fundamental challenge of monitoring CFGs. The *first* set of a non-terminal A, denoted *first(A)*, is the set of all terminals t such that the sub-strings which reduce to A may possibly begin with t. The follow set, denoted *follow(A)*, is the set of terminals which follow the strings which reduce to A. These terminals signify a reduction by A.

A reduction is the step whereby a right hand side of a production, γ, is replaced by the left hand side non-terminal in the production. For example, if we have the string *aaabbb*, and we are using our example grammar, G , we can perform a reduction with $\gamma = \epsilon$ resulting in *aaaAbbb*. We can then perform another reduction with $\gamma = aAb$ resulting in *aaAbb*, eventually we reach *aAb*, which reduces to A (and because A is the start symbol, *aaabbb* must be in $L(G)$, where $L(G)$ represents the language derived by G).

5.2 CFG Monitoring Algorithms

We developed the CFG monitoring algorithms based on existing parsing algorithms. *Action* and *goto* tables are generated from the given CFG pattern and used to advance the monitor at runtime according to the observed events. Moreover, the CFG monitor is unaware of relevant parameters since they are handled by the underlying MOP framework. This greatly simplifies the monitoring algorithm. We next introduce the monitor algorithms in more detail.

5.2.1 CFG Simplification

The CFG plug-in first applies some standard simplifications to the given grammar (Hopcroft et al, 2001). The first step of simplification is the removal of non-generating nonterminals (A is non-generating if $\forall s \in \Sigma^*$, s cannot be reduced to A in one or more reductions). The next step in the simplification process is the removal of nonterminals which are unreachable from the start symbol (A is unreachable from the start symbol if there is no string γ that contains A and reduces to the start symbol in any number of steps). The last step removes ϵ-productions from the grammar. After ϵ-productions have been

```
1  globals action_table, goto_table
2  initialize stack.push(initial_state)
3  procedure monitor(event, stack)
4    locals state, state', stack', A
5    state ← stack.top()
6    while (true) {
7      switch (action_table[state,event].action_type) {
8      case shift :
9        state' ← action_table[state, event].next_state
10       if (state' = error) {
11         pattern failure
12         break while
13       }
14       stack.push(state')
15       if (action_table[state', $].action_type = reduce) {
16         stack' ← stack.copy()
17         monitor($, stack')
18       }
19       break while
20     case reduce :
21       stack.pop(action_table[state, event].pop)
22       A ← action_table[state, event].non_terminal
23       state' ← stack.top()
24       stack.push(goto_table[state', A])
25       break switch
26     case accept :
27       pattern match
28       break while
29     }
30   }
```

Fig. 3 CFG Monitoring Algorithm with Stack Copying.

removed from G, resulting in G', $L(G') = L(G) - \epsilon$.[9] A monitor matching the empty event trace has little utility, so we feel this is a fair compromise.

5.2.2 Tables and Monitoring

After simplification, the tables are generated for a deterministic push-down automaton (DPDA), that is, a deterministic finite automaton with a stack, which is to be used as a monitor. The algorithm first adds a production to

[9] The remaining productions are restructured to account for the removal of ϵ-productions without changing the recognized language, other than as mentioned.

	Action					Goto	
	$	acquire	release	begin	end		S
0	shift(error)	shift(14)	shift(error)	shift(16)	shift(error)	0	9
1	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	1	error
2	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	2	error
3	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	3	error
4	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	4	error
5	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	5	error
6	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	6	error
7	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	7	error
8	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	8	error
9	accept	shift(15)	shift(error)	shift(17)	shift(error)	9	error
10	shift(error)	shift(15)	shift(3)	shift(17)	shift(error)	10	error
11	shift(error)	shift(15)	shift(error)	shift(17)	shift(4)	11	error
12	shift(error)	shift(15)	shift(7)	shift(17)	shift(error)	12	error
13	shift(error)	shift(15)	shift(error)	shift(17)	shift(8)	13	error
14	shift(error)	shift(14)	shift(1)	shift(16)	shift(error)	14	10
15	shift(error)	shift(14)	shift(5)	shift(16)	shift(error)	15	12
16	shift(error)	shift(14)	shift(error)	shift(16)	shift(2)	16	11
17	shift(error)	shift(14)	shift(error)	shift(16)	shift(6)	17	13

Table 2 LALR(1) Tables for SafeLock

introduce $. If the start symbol of the original grammar was S, it adds a production $S' \rightarrow S\$$. The next step is generating the *canonical LR(1) (or LALR(1)) collection*. The collection consists of *collection items*; each item represents a state in the automaton. A collection item is a set of productions with a marker, *, for the current position in the right hand side, augmented with a look-ahead. For example, a possible collection item for the simple HasNext pattern $S \rightarrow next\ next$ is $[\{S' \rightarrow *\ S\ \$, \$\}, \{S \rightarrow *\ next\ next, \$\}]$, another is $[\{S \rightarrow next\ *\ next, \$\}]$. In both of these collection items, the look-ahead is $. The collection item is first created for the start production, $S' \rightarrow S\$$. All productions for S are added to the state with * at the beginning of the right hand side, as can be seen in our example collection item. If there is a production for S such that the first symbol is a nonterminal, A, all the productions of A will also be added, with * at the beginning. This process is transitively closed. The next collection items are generated by advancing * in the production, and then taking the transitive closure for any nonterminals immediately following *. Once the collection is created, the tables are generated by treating each item as a state. The productions in the item are considered for each alphabet symbol (including $). If the marker appears in front of said terminal, a *shift* action is generated. The algorithm decides which collection item to shift to by looking for the collection item where there is a production with the same right hand side as the production that caused the shift action, but with * advanced one position. For example, the next collection from $[\{S \rightarrow next\ *\ next, \$\}]$ would be $[\{S \rightarrow next\ next\ *, \$\}]$. *Shift* actions can never be generated for $; the algorithm disallows it. The handling of *shift* actions by the parsing algo-

rithm can be seen below. If, however, there is a production in the item such as $[\{S \rightarrow next\ next\ *, \$\}]$, where the marker appears at the end, and the terminal in question is the look-ahead, a *reduction* action is generated. More explanation of this algorithm can be found in (Aho et al, 1986).

When a new event arrives, the monitoring algorithm must decide how to modify the stack. The tables are given in a generic intermediate form, which is converted by the Java shell into two Java arrays.

Pseudo-code for our monitoring algorithm is given in Figure 3. The significance of lines 15-18 is explained Section 5.2.4. An entry in *action_table* specifies, via the *action_type*, the type of action: *shift*, *reduce*, or *accept*. Each type of action also requires additional information in order for the algorithm to process said action. An entry in *goto_table* simply identifies the next state for the DPDA. Table 2 shows a parse table as it would be used by the algorithm. This is the LALR(1) table for the first SafeLock grammar given in Section 1. We show the LALR(1) table because the LR(1) table has 50 states, and the tables can be used interchangeably by the algorithm.

The *shift* action entry contains the next state for the DPDA in the *next_state* field (in parentheses in the shift actions in Table 2). A *shift* action simply pushes the next state on the stack, if the next state is *not* the error state (lines 10-14). If, however, the table indicates that the next state *is* the error state, the algorithm reports a pattern fail and breaks *without* touching the stack (lines 11-12). This allows the algorithm to continue to find more pattern failures. After a successful *shift* action, the while loop is broken, allowing execution of the monitored program to continue until the next relevant event (line 19).

The *reduce* action is more complicated. The field *non_terminal* describes which non-terminal (A) the production $A \rightarrow \gamma$ reduces to (the first field in the reduce actions Table 2), while the field *pop* denotes how many states to pop from the stack ($|\gamma|$) (the second field in the reduce actions in Table 2). The reduction proceeds by popping the specified number of states from the stack and consulting *goto_table* to decide the next state (lines 20-25). The state used for indexing *goto_table* is not the current state, but rather the state at the top of the stack after the specified number of states has been popped (line 21). An indeterminate number of reductions can happen in a row, but there *must* be *shift* at the end of the reduction sequence before the algorithm can terminate for a given event. The reductions happen before the *shift* to simulate the look-ahead of one token specified by the 1 in LR(1).

The *accept* action, which directs the DPDA to signal a pattern match, has no special fields, as no more information is necessary (lines 26-28).

5.2.3 The LALR(1) Optimization

LALR(1) table generation is a standard modification of the LR(1) table generation algorithm (Aho et al, 1986). LALR(1) tables are constructed the same way as LR(1) tables, save that states corresponding to collection items with the same *core* are merged as they are discovered. The core of a collection item

is that item with the look-aheads removed. This means that LALR(1) tables can be no larger than LR(1) tables, but may be considerably smaller.

However, this process may result in reduce-reduce conflicts (states where a reduction to two or more different nonterminals with the same look-ahead may occur) that do not exist when the LR(1) construction method is used. Shift-reduce conflicts (states where a shift with a given token, a, and a reduction with a as the look-ahead are possible) cannot be introduced because they require that there be two productions in a given collection item such that one has the position marker in front of a terminal t (meaning that t should be shifted), while another production has the marker at the end of the production, with t as the look-ahead (meaning that we should reduce to the left hand side when t is seen). Obviously the conflict must occur *before* merging, because if the shift inducing production were in a state s_0, while the reduce inducing production where in s_1, s_0 and s_1 would have different cores, and not be merged. To see how reduce-reduce actions may be introduced, consider the two collection items: $[\{A \rightarrow a, \$\}, \{B \rightarrow a, a\}]$ and $[\{A \rightarrow a, a\}, \{B \rightarrow a, \$\}]$ Before merger, no conflict exists. Looking at the first collection item, there is no conflict between the two productions because it says to reduce to A only when the look-ahead is \$, and B only when the look-ahead is a. The collection item after merger, however, is: $[\{A \rightarrow a, \$\}, \{B \rightarrow a, a\}, \{A \rightarrow a, a\}, \{B \rightarrow a, \$\}]$, which contains two reduce-reduce conflicts (one on look-ahead a, the other on \$). The only way to check if such a conflict is introduced by the LALR(1) merger is to generate LR(1) tables to see if there is still a reduce-reduce conflict.

5.2.4 Handling the End of Trace

The (LA)LR(1) algorithm assumes that the string of terminals to be evaluated is *completely known ahead of time*. Thus, it knows where the end of the string (denoted as \$) is. This is important because some reductions happen with the \$ symbol as the look-ahead, and the accept action can *only* be recognized when the next input is \$. To be consistent with our notion of monitoring, it must be possible to consider the trace prefix at a given point in a run of a program as an *entire* trace. The algorithm must then assume \$ after every event.

Our implementation of the algorithm attempts to reduce with \$ as the look-ahead after *every* valid *shift* (lines 15-18). The problem with reducing with \$ as the look-ahead where possible is that all state of the current trace evaluation is lost. This means that the monitor could only accept the minimal trace that matches the CFG pattern if no special care were taken.

Since our notion of monitoring reports pattern matches for every current trace that matches the pattern[10], one possible option is to *copy* the stack before we perform any reductions with \$ as the look-ahead.

This copying ensures that the stack is intact for the next, and subsequent events, allowing for multiple pattern matches. For example, consider the language denoted by the regular expression ab^*. While we would suggest using the

[10] This is irrespective of suffix matching which actually generates multiple monitors.

ERE plug-in for such a language, it is a clear example to illustrate the effect of copying. With no copying the algorithm would accept for only a. Because it popped during the pattern match phase, if it sees a b it will report failure. With copying it will report success for a, and then success again on the input of b, and for any subsequent input of b. An important optimization to copying is to only copy the stack if there is a reduction with $ as the look-ahead, rather than blindly for every *shift* operation. This optimization will not help ab^*, but it will help for many other languages. In fact, for $\{a^n b^n | n \in \mathbb{N}\}$, only one copy is necessary no matter how long the input. Any grammar accepting unbounded repetition at the end of the pattern (like ab^*), will require copying on each input of the repeated character.

Our experience with stack copying led us to an important observation: when there is a reduction with $ as the look-ahead, acceptance is *guaranteed*. That is to say, there is never a situation in which there is a reduction with $ as the look-ahead that results in a parse failure. This is a consequence of the parse table generation algorithm, and Section 5.4 covers the correctness of this notion, which we refer to as *guaranteed acceptance*. Using guaranteed acceptance to accept whenever a reduction with $ as the look-ahead is possible is another alternative for matching all good prefixes of a pattern. Guaranteed acceptance is always correct. We maintain the discussion on stack copying because the proof of the stack copying algorithm is easier to understand, and because it is an interesting, though less practical, alternative to guaranteed acceptance.

5.3 CFG-plug-in Implementation

The CFG plug-in allows the user to specify a number of events and a CFG specifying allowable event traces. The events become the terminals of the CFG, i.e., Σ. The translation steps from specification to working Java code gradually transform the specification into AspectJ join points (the events) and aspects (the synthesized monitors), which are then woven into the original application using any off-the-shelf AspectJ compiler.

5.3.1 Suffix Matching with CFGs

As described in Section 4, several features are needed for monitors to support optimized suffix matching.

The first is identification of monitor creation events[11]. As already mentioned, monitor creation events are events which, when encountered as the *first* event in a trace, would not lead to an immediate failure. For CFGs this would imply an event that can begin a sub-string which reduces to the start symbol. This is the same as the definition of *first* set as given earlier. Thus, the monitor creation events for the CFG plug-in are those events which are in *first(S)*, where S is the start symbol for the grammar.

[11] Though, as mentioned, this is also necessary for total matching.

Additionally, it is necessary to define a hash encoding for CFG based monitors because our suffix matching algorithm uses a HashSet to find monitors with *potentially* equivalent states quickly. We decided that two simple defining aspects of CFG based monitors are stack depth and the current state of the monitor (the top of the stack). We chose to xor them together (a broadly used operation for combining two binary quantities into one quantity representative of the two in the same number of bits) because the hashCode method must return an integer. Lastly, we need an equality method (to resolve collisions) defining when two CFG based monitors have *actually* equivalent states. Two CFG monitors can only be equal iff they have the same stack contents. It will be fairly rare for two proper CFG monitors to be equivalent, as they do not have finite state like the other logic plug-ins of MOP. Thus, it is important for failed equality testing to be fast. Because of this, we check to see if two monitors have the same stack depth before beginning element-wise comparisons.

5.4 Proofs of Correctness

We next prove the correctness of the proposed CFG monitoring algorithms.

First, we prove the online monitoring algorithm for CFG using stack copying correct. We achieve this by showing that our algorithm detects pattern failures and pattern matches of the observed trace in the same way as the ASU parsing algorithm(Aho et al, 1986), as given in Figure 4.

Theorem*: For every finite prefix of a (possibly infinite) program trace*[12] *and a CFG pattern, the MOP algorithm will notify a failure of the pattern if the ASU algorithm would notify a parse failure due to a bad token, and pattern match if ASU would notify success, given that prefix as total input.*

Proof: First, we construct a new parsing algorithm, as shown in Figure 5. This new algorithm can be proved equivalent to the one in Figure 4 as follows. The major difference between these two algorithms is that the pointer (*ip*) increment is moved to the outer loop in Figure 5. This change does not affect the behavior of the algorithm:

1. For a shift action, both algorithms carry out the same operation except that Figure 4 increases the pointer and continues to the next action, while Figure 5 breaks the inner loop, increases the pointer in the outer loop, and then continues to the next action. Both are equivalent.
2. For reduction, Figure 5 chooses to stay in the inner loop, which is identical to Figure 4, and continues the loop without increasing the pointer.
3. For acceptance (pattern match in monitors), both algorithms are identical.

Now we can prove the correctness of the monitoring algorithm in Figure 3 by comparing it with the modified parsing algorithm in Figure 5.

The major difference distinguishing the monitoring algorithm from ASU is that the former has to wait for the next event extracted from the execution of

[12] Each prefix is an $\mathcal{E}$-trace at a given point in a program as per Definition 1.

```
 1 globals stack, ip, action_table, goto_table
 2 initialize stack.push(initial_state), ip ← 0
 3 procedure parse()
 4   locals state, state', a, A
 5   while (true) {
 6     state ← stack.top()
 7     a ← get_token_at(ip)
 8     switch (at[state, a].action_type) {
 9     case shift :
10       state' ← action_table[state, a].next_state
11       if (state' = error) {
12         report_error
13         advance ip
14         continue while
15       }
16       stack.push(state')
17       advance ip
18       continue while
19     case reduce :
20       stack.pop(action_table[state, a].pop)
21       A ← action_table[state, a].non_nonterminal
22       state' ← stack.top()
23       stack.push(goto_table[state', A])
24       continue while
25     case accept :
26       accept
27       return
28     }
29   }
```

Fig. 4 ASU Algorithm.

the monitored program while the latter can actively retrieve the next token, which is handled in the outer loop in Figure 5. Therefore, we only need to prove that the *monitor* procedure in Figure 3 produces the same result as the inner loop in Figure 5, given the same state and event to process.

It is straightforward to compare both pieces of code: the only difference between them is the *stack copying* (lines 14-17) in Figure 3. It is needed because we wish to continue parsing *after an accept*, and because we can never actually see $ as an event. We copy the stack after a shift and check for actions with $ as the input. The only actions possible on this recursive call are reduce and accept because $ can never be shifted[13]. Due to this, the recursion is

[13] This is a property of the CFG parsing table generation algorithm, which we use without proof. It is obvious, however, because $ is not a part of the original grammar.

```
 1 globals stack, ip, action_table, goto_table
 2 initialize stack.push(initial_state), ip ← 0
 3 procedure parse()
 4   locals state, state', a, A
 5   while (true) {
 6     a ← get_token_at(ip)
 7     while (true) {
 8       state ← stack.top()
 9       switch (at[state, a].action_type) {
10       case shift :
11         state' ← action_table[state, a].next_state
12         if (state' = error) {
13           report_error
14           break while
15         }
16         stack.push(state')
17         break while
18       case reduce :
19         stack.pop(action_table[state, a].pop)
20         A ← action_table[state, a].non_nonterminal
21         state' ← stack.top()
22         stack.push(goto_table[state', A])
23         continue while
24       case accept :
25         accept
26         return
27       }
28     }
29     advance ip
30   }
```

Fig. 5 Modified ASU Algorithm.

always bounded at depth one. This is the major difference between the MOP and ASU algorithms. Because $ can never be an event, we must speculatively guess the end of input after every symbol seen. The recursive call must happen iff there is a valid reduction with $ as the look-ahead. Because the algorithm repeats until a shift action, error, or accept happens, we ensure that, if the recursive call happens, it must happen after the processing of each input[14]. Cloning the stack allows us to reduce and accept, while still maintaining the original stack to continue monitoring events as if the end of input had *not*

[14] Accept need not be considered because it can only happen when the input is $, which only occurs during a recursive call.

```
1  globals action_table, goto_table
2  initialize stack.push(initial_state)
3  procedure monitor(event, stack)
4    locals state, state', stack', A
5    state ← stack.top()
6    while (true) {
7      switch (action_table[state,event].action_type) {
8      case shift :
9        state' ← action_table[state, event].next_state
10       if (state' = error) {
11         pattern failure
12         break while
13       }
14       stack.push(state')
15       if (action_table[state', $].action_type = reduce) {
16         pattern match
17       }
18       break while
19     case reduce :
20       stack.pop(action_table[state, event].pop)
21       A ← action_table[state, event].non_terminal
22       state' ← stack.top()
23       stack.push(goto_table[state', A])
24       break switch
25   }
```

Fig. 6 CFG Monitoring Algorithm with Guaranteed Acceptance.

been seen. Thus, this change is equivalent to the ASU algorithm in terms of language recognition because both possibilities (the arrival or non-arrival of $) are explored. That is, the MOP algorithm will report accept for a given prefix if ASU would, given that prefix as its total input, and it, additionally, retains enough state to continue parsing future (longer) traces. Violation is handled identically in both algorithms. □

We next prove the correctness of our online monitoring algorithm for CFG using the notion of *guaranteed acceptance*. It is achieved by showing that a reduction with $ as the look-ahead must result in accept with $ as look-ahead[15] in one or more reductions. Figure 6 shows the algorithm modified to take advantage of guaranteed acceptance. Note that there is no longer an accept case because accept is discovered in the shift case, on lines 15-16.

[15] Note that this is the only look-ahead ever possible for accept.

Theorem*: If the action table entry for a given state specifies reduction with* $ *as look-ahead, the stack copying processes must lead to an acceptance. That is to say, in one or more reductions with* $ *as the look-ahead, an accept action must occur with* $ *as the look-ahead.*

Proof: From the canonical LR(1) construction algorithm (Aho et al, 1986), we know that a given non-terminal B can only be reduced to with look-ahead terminal a, if there exists some production $C \rightarrow \gamma_0 Ba\gamma_1$ or a sequence of productions $C \rightarrow \gamma_0 B_0 a\gamma_1$, $B_0 \rightarrow \gamma_2 B_1$, $B_1 \rightarrow \gamma_3 B_2$, ... $B_n \rightarrow \gamma_{n+2} B$. Note that the sequence may contain cycles of one or more production, such as a production $C \rightarrow bC$, but there must be a finite amount of "cycle reductions" because there is a finite amount of input, and the CFG simplification we perform removes non-generating nonterminals[16].

In the algorithm, $ is treated as a special terminal that exists in only one production, the start production $S' \rightarrow S\$$, where S is the original start symbol of the grammar. This production is added by the algorithm and is the only existence of $ in the grammar. Intuitively, when the contents of the parse stack *correspond* to S, the algorithm accepts. This means that if we can reach a stack containing only state corresponding to S through one or more reductions with $ as the look-ahead, we can accept. The original ASU algorithm and our version of the algorithm with stack copying perform all of these reductions.

As a result of the two facts above, a reduction with $ as look-ahead, for non-terminal A, can only occur in the table if there is some sequence of productions $S \rightarrow A_0\$$, $A_0 \rightarrow \gamma_0 A_1$, $A_1 \rightarrow \gamma_1 A_2$, ... $A_n \rightarrow \gamma_n A$, or the production $S \rightarrow A\$$. If we have $S \rightarrow A\$$, then we can obviously accept on the reduction to A because this will result in a stack with only contents corresponding to A ($A = S$ from above), which is the accept condition, or there will be some finite number of cycles with non-terminal A that eventually leads to a stack containing only contents corresponding to A because there must be a finite amount of cycle reductions. If, however, we have the sequence of productions, we can still accept because there is a sequence of reductions from A to $S \rightarrow A_0\$$ given by the sequence of productions. Again, even if the sequence contains cycles, eventually acceptance must be reached because there must be a finite amount of cycle reductions. □

6 Evaluation

We evaluated the JavaMOP CFG plug-in and compared its performance to PQL and Tracematches on the DaCapo benchmark suite (Blackburn et al, 2006). We used the LR(1) tables, with the lazy algorithm, and suffix matching. We feel this gives a worst case. LR(1) tables are of greater or equal size, so they cannot be faster than the LALR(1) tables. The lazy mode has the potential to be slower because it continues to modify the stack after a failure, while the normal mode does not. Suffix matching is obviously slower than total trace

[16] It is clear that only non-generating nonterminals could have infinite cycles, because the table generation algorithm works bottom up, and chooses shift over reduce on conflict.

matching because it creates one total match monitor for every suffix of the trace.

6.1 Experimental Settings

Our experiments were carried out on a machine with 1.5GB RAM and Pentium 4 2.66GHz processor. The operating system used was Ubuntu Linux 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs (Blackburn et al, 2006): antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd, and xalan. The provided default input was used together with the -converge option to execute the benchmark multiple times until the execution time falls within a coefficient of variation of 3%. The average execution time of six iterations after convergence are then used to compute the runtime overhead. Therefore, Table 3 percentages should be read "±3" (meaning negative numbers are possible).

6.2 Properties

The following general properties borrowed from (Bodden et al, 2007) were checked in the evaluation:

- HashMap: An object's hash code should not be changed when the object is a key in a HashMap;
- HasNext: For a given iterator, the hasNext() method should be called between all calls to next();
- SafeIterator: Do not update a Collection when using the Iterator interface to iterate its elements.

We also defined three new properties to showcase the power of the CFG plug-in; they are all properly context-free:

- ImprovedLeakingSync: The original LeakingSync specified in (Bodden et al, 2007) *only* allows synchronized accesses to synchronized collections. This causes spurious failures because the synchronized methods call the unsynchronized versions. Our improved version allows calls to the unsynchronized methods so long as they happen within synchronized calls.
- SafeFileInputStream: SafeFileInputStream is a modification of our SafeLock property from Figure 2. It ensures that a FileInputStream is closed in the same method in which it is created.
- SafeFileWriter: SafeFileWriter ensures that all writes to a FileWriter happen between creation and close of the FileWriter, and that the creation and close events are matched pairs.

More properties have been checked in our experiments; we chose the first three regular-language-based properties (HashMap, HasNext, and SafeIterator)

to include in this paper because they generate a comparatively larger runtime overhead. We excluded those with little overhead in JavaMOP. For every property, we provide overhead percentages for JavaMOP, as well as PQL and Tracematches where possible. We run the JavaMOP monitors in suffix matching mode; the decentralized indexing of monitors was used in all the experiments (see (Chen and Roşu, 2007)). We chose the AspectJ compiler 1.5.3 (AJC) in the evaluation to compile the JavaMOP generated monitoring AspectJ code. Guaranteed acceptance and stack copying have the same performance on each of these patterns because none of the properties is able to generate more than one pattern match from a given parameter instance, meaning that the number of stack copies is very minimal. Additionally, the properties have been written in a method that ensures minimal stack size (such as using left recursion instead of right recursion). For Tracematches we used the most recent published version from (Bodden et al, 2008).

	HashMap			HasNext			SafeIterator		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	3	6	0	1	2	3	2	82	0
bloat	14	9	-2	1112	5929	2452	627	8694	11258
chart	-1	1	-1	-1	3	0	2	50	11
eclipse	0	1	1	0	2	-1	-2	1	2
fop	3	2	0	0	2	-1	-1	24	5
hsqldb	0	3	15	0	6	15	0	78	17
jython	0	23	15	0	0	13	0	12	16
luindex	1	8	1	-2	93	2	3	181	9
lusearch	1	1	8	-1	59	9	4	132	34
pmd	-1	0	3	191	1870	52	178	1334	175
xalan	0	5	1	0	0	2	1	53	10
	ImprovedLeakingSync			SafeFileInputStream			SafeFileWriter		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	1	N/E	N/E	3	113	-1	2	22	N/E
bloat	13	N/E	N/E	1	128	0	27	97	N/E
chart	4	N/E	N/E	0	29	1	0	37	N/E
eclipse	1	N/E	N/E	-2	3	0	-2	1	N/E
fop	1	N/E	N/E	-2	58	-1	-2	47	N/E
hsqldb	1	N/E	N/E	1	280	21	2	95	N/E
jython	41	N/E	N/E	0	937	12	1	crashes	N/E
luindex	1	N/E	N/E	-1	233	6	0	33	N/E
lusearch	2	N/E	N/E	-1	137	7	0	49	N/E
pmd	36	N/E	N/E	-1	547	1	-2	658	N/E
xalan	3	N/E	N/E	-1	90	3	-2	164	N/E

Table 3 Average percent runtime overhead for JavaMOP CFG (MOP), PQL, and Tracematches (TM) (convergence within 3%); N/E means "not expressible".

Property	HashMap	HasNext	SafeIterator	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
antlr	0	0	1990	8472	0	0
bloat	361519	143103032	75944328	5587905	259	385
chart	8773	6819	569345	634260	0	0
eclipse	20888	3252	32759	74630	930	0
fop	17265	281	49959	182407	12	0
hsqldb	0	0	0	0	0	0
jython	443	106	177554	23969673	544	0
luindex	9615	28140	82162	1559386	1114	0
lusearch	416	0	405428	1291992	0	0
pmd	11354	33294563	25476563	26291289	10	32
xalan	124155	0	1009649	5146036	13604	0

Table 4 Number of events handled by JavaMOP

6.3 Results

Table 3 shows the percent overheads of JavaMOP using the CFG plug-in, PQL, and Tracematches. N/E refers to specifications that were not expressible. Negative numbers can be attributed both to the 3% noise in the measurements and instruction cache layout changes due to the weaving process. Tracematches is unable to support ImprovedLeakingSync because the property is truly context-free. PQL is also unable to support it because it requires events corresponding to the beginning and end of synchronized method calls, and PQL can only trigger events on the end of method calls. Tracematches cannot support SafeFileWriter because it is a pure context-free specification. However, Tracematches *can* support SafeFileInputStream because it has the ability to access call stack depth via the cflowdepth pointcut term, which is provided only by the ABC compiler for AspectJ.

Over one run of the entire DaCapo benchmark suite, more than 355 *million* events (Table 4) were triggered. Tracematches has the same number of events throughout the tests because it uses the same instrumentation technique as JavaMOP. We had no good method to obtain the number of events generated in PQL; we assume it was less because PQL performs a static optimization which removes unnecessary optimization points. It is interesting to note that in the cases of HasNext with antlr, lusearch, and xalan that there are no events, despite the fact that these three benchmarks have events for SafeIterator. The reason for this is that the SafeIterator instruments Collection.remove, so it is possible for SafeIterator to have events in programs with no actual Iterators.

The average overhead of JavaMOP over 45 program/property pairs that actually generate events is 50%. There are two considerations here, however: (1) we chose specifically those properties that generated the largest overheads (HasNext and SafeIterator in bloat), (2) when the two largest overheads are removed, the average over the remaining 43 pairs drops to a very reasonable 12%. Further, the average JavaMOP overhead for properties expressible in PQL that generated events was 61% over 36 pairs, while PQL's overhead on these same properties was 583%. Similarly, for Tracematches expressible properties that generated events, JavaMOP's overhead was 64% over 33 pairs, while Tracematches was 414%. Tracematches, PQL, and JavaMOP all feature the same two pairs which have extremely large overhead compared to the me-

Property	Original	HashMap	HasNext	SafeIterator
antlr	2.3 / 10.1	2.0 / 10.6	1.8 / 10.6	2.0 / 10.8
bloat	5.6 / 8.9	6.9 / 8.9	5.9 / 8.7	541.0 / 10.6
chart	20.1 / 11.3	20.8 / 11.4	17.0 / 11.3	20.7 / 11.5
eclipse	27.0 / 22.1	30.7 / 22.2	27.4 / 22.1	28.6 / 22.3
fop	12.3 / 9.1	13.2 / 9.2	10.9 / 9.0	10.2 / 9.1
hsqldb	80.8 / 7.6	80.2 / 7.6	76.4 / 7.5	77.5 / 7.6
jython	3.9 / 19.0	4.1 / 19.0	3.8 / 19.0	3.9 / 19.1
luindex	4.2 / 6.9	4.0 / 7.0	4.6 / 6.9	4.7 / 7.1
lusearch	5.2 / 6.2	5.2 / 6.3	5.7 / 6.2	5.3 / 6.3
pmd	22.0 / 8.6	22.3 / 8.7	24.0 / 8.6	888.1 / 8.9
xalan	21.7 / 10.2	23.8 / 10.5	26.2 / 10.2	29.1 / 10.3
Property	Original	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
antlr	2.3 / 10.1	2.1 / 10.7	2.4 / 10.7	2.2 / 10.7
bloat	5.6 / 8.9	7.9 / 10.0	5.0 / 8.9	5.6 / 8.9
chart	20.1 / 11.3	17.0 / 11.3	17.8 / 11.3	16.4 / 11.3
eclipse	27.0 / 22.1	28.9 / 22.1	30.7 / 22.1	27.1 / 22.1
fop	12.3 / 9.1	14.6 / 9.2	11.9 / 9.0	12.0 / 9.0
hsqldb	80.8 / 7.6	87.2 / 7.5	78.2 / 7.5	79.3 / 7.5
jython	3.9 / 19.0	4.0 / 19.2	4.0 / 19.1	3.6 / 19.1
luindex	4.2 / 6.9	5.6 / 7.0	4.2 / 6.9	4.6 / 6.9
lusearch	5.2 / 6.2	5.8 / 6.4	5.6 / 6.3	5.7 / 6.3
pmd	22.0 / 8.6	22.2 / 8.8	24.2 / 8.6	22.9 / 8.6
xalan	21.7 / 10.2	24.4 / 10.3	22.0 / 10.3	26.5 / 10.2

Table 5 Maximum memory usage in MB (Maximum Heap Memory Usage) / (Maximum Non-Heap Memory Usage).

dian (HasNext and SafeIterator in bloat). When these two pairs are removed from the three averages, the average overhead for JavaMOP with respect to PQL expressible properties is 12%, while PQL still weighs in at 199%. Tracematches is comparable to JavaMOP, with JavaMOP and Tracematches both at 12%. Since Tracematches does not support the full generality of (deterministic) context-free grammars, we view comparable performance to Tracematches as favorable to our approach, especially given that, in the overall data set, our average overhead is over 8 times lower than Tracematches' overhead.

The largest overheads seen, across all three systems, are for SafeIterator and HasNext in bloat. This is due to bloat's extensive use of iterators. Bloat is a bytecode optimizer, which uses iterators to process bytecode. PQL and Tracematches perform worse on SafeIterator than they do on HasNext, while our performance is the opposite. The reason for this is that HasNext creates a far larger number of monitors in JavaMOP because it creates a monitor for every call to next, while SafeIterator only creates monitors on a call to create. The pattern for SafeIterator, however, is more complex. This shows that JavaMOP has, relatively speaking, more overhead in generating and handling the monitor set for suffix matching than it does in matching the pattern, while PQL and Tracematches overheads are more affected by the complexity of the pattern. Note that JavaMOP with CFGs far outperforms both PQL and Tracematches on these 2 program/property pairs.

	HashMap	HasNext	SafeIterator	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
LR(1) states	6	5	15	19	20	18
LALR(1) states	6	5	15	15	8	11

Table 6 Comparison of LR(1) and LALR(1) tables.

SafeFileInputStream is an interesting case: it is required to match the begin and end of methods. Instrumenting the begin and end of *every* method would be atrociously slow, however. We perform a static analysis which finds those methods in which FileInputStream's are actually used. Then, we instrument only those methods for begin and end. Because Tracematches, also, is pointcut based, we are able to perform the same optimization for Tracematches, so the numbers shown are with the optimization enabled. PQL is not pointcut based so the optimization cannot be applied; however, the PQL property does not match begins and ends of methods (recall: PQL can only match the ends), so this is not an issue. In PQL we specify SafeFileInputStream by using an interesting PQL-specific feature called within. The idea of within is that a property matches only *within* a given method or methods matching a particular pattern (in the case of SafeFileInputStream we use the pattern _._ which specifies all methods of all classes). Additionally, PQL will only instrument the same methods that JavaMOP and Tracematches instrument because within only instruments methods which can generate relevant events.

The memory overhead is reasonable in our experiments: overall, it is 33% on average with a 4% median (see Table 5 for a pair-wise breakdown). There are two extreme cases of memory overhead caused by JavaMOP monitors: bloat-SafeIterator and pmd-SafeIterator. Our investigation shows that both programs, bloat and pmd, make intensive use of vectors, and create numerous iterators to do computation over the vectors throughout the execution. Note that every creation of the iterator leads to the creation of a monitor instance for SafeIterator using our technique. Hence, a huge number of monitor instances were created in these two benchmarks. While the iterator object is usually used in a small scope and then released, the vectors are not released until the end of the execution, preventing the removal of the created monitor instances. In other words, all the monitor instances created during the execution of bloat and pmd were kept alive until the execution ended, resulting in the observed massive memory usage. On the contrary, we can see that a large number of monitors were also created for bloat-HasNext and pmd-HasNext but with much less memory overhead. HasNext has one monitor created for every iterator object, and when the iterator is released, the corresponding monitor will also be removed. Since most iterators were released shortly after creation, only a few monitors existed at the same time during the execution resulting in much lower memory overhead. Compared with the results of Tracematches (Avgustinov et al, 2007), we believe there is still some room for improvement with regard to memory usage in our approach. The memory overhead of our approach does not cause unnecessary loss of performance during the evaluation, indicating that it is not a bottleneck for the efficiency of monitoring.

Table 6 shows a comparison of parse table size between LR(1) and LALR(1) in terms of number of states. No reduce-reduce conflicts were introduced in any of our properties, and the space savings can be significant. Although, in either case, the tables are small enough that the size difference has no effect on runtime. It is interesting to note that the three regular language properties see no savings from LALR(1).

Our experiments with guaranteed acceptance found no benefit to the patterns we tested because, as mentioned, each pattern can only accept once per parameter instance, and the stack depth is kept to a minimum by using careful grammar design. Not all properties, however, can or should be written in such a way that only one acceptance can be generated per parameter instance. In extreme cases, such as patterns that feature unbounded repetition at the end of the pattern, such as a^*, guaranteed acceptance can provide a *lower asymptotic complexity*. Consider the grammar $S \rightarrow \epsilon|aS$ which monitors a^*. For each a that arrives, with guaranteed acceptance a *constant time*[17] step of adding a to the stack and accepting occurs. With stack copying, the stack must be copied on each arrival of a. This copying, however, will take time linear in the amount of times a has been previously seen. Thus, with guaranteed acceptance, the monitoring time is linear in the number of a events in the program run while stack copying is quadratic.

7 Conclusions and Future Work

We implemented a CFG logic plug-in for JavaMOP using a modified LR(1) parsing algorithm. We also implemented an optimization of table generation, which uses the LALR(1) state merging technique, leading to smaller tables, but may result in extra reduce-reduce conflicts. Our first modification to the algorithm is based on the novel idea of *copying* the stack in order to "predict" a possible reduction with $ (end of string) as a look-ahead without destroying the state of the monitor. An important optimization and simplification possibility is *guaranteed acceptance*, wherein the algorithm accepts when a reduction with $ as look-ahead is possible; this saves the copying operation, which can take arbitrarily long to perform, since the stack is unbounded. We showed, empirically, that our algorithm is efficient and faster than the state-of-the-art for monitoring CFG properties. We also extended JavaMOP with suffix matching in order to fairly compare JavaMOP with PQL and Tracematches. Tracematches, however, cannot handle arbitrary context-free patterns.

We have also begun work on a logic called PtCaRet as another specification formalism to support structured specifications. PtCaRet is past time linear temporal logic (PTLTL) with calls and returns. It is a super set of PTLTL that provides operators which apply only on a function/method local level.

[17] It is constant except when the stack size must be grown.

References

Aho AV, Sethi R, Ullman JD (1986) Compilers, principles, techniques, and tools. Addison-Wesley, pages 215-246

Allan C, Avgustinov P, Christensen AS, Hendren LJ, Kuzins S, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J (2005) Adding trace matching with free variables to AspectJ. In: OOPSLA'05, ACM, pp 345–364

Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhotak J, Lhotak O, de Moor O, Sereni D, Sittampalam G, Tibble J (2005) ABC: an extensible AspectJ compiler. In: AOSD'05, ACM, pp 87–98

Avgustinov P, Tibble J, de Moor O (2007) Making trace monitors feasible. In: OOPSLA'07, ACM, pp 589–608

Barringer H, Rydeheard D, Havelund K (2008) Rule systems for run-time monitoring: from eagle to ruler. J Logic Computation pp exn076+

Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B (2006) The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA'06, ACM, pp 169–190

Bodden E (2005) J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University

Bodden E, Hendren L, Lhoták O (2007) A staged static program analysis to improve the performance of runtime monitoring. In: ECOOP'07, LNCS, vol 4609, pp 525–549

Bodden E, Lam P, Hendren L (2008) Tracematches Benchmarks. `http://abc.comlab.ox.ac.uk/tmahead`

Chaudhuri S, Alur R (2007) Instumenting C programs with nested word monitors. In: Model Checking Software(SPIN'07), LNCS, vol 4595, pp 279–283

Chen F, Roşu G (2003) Towards monitoring-oriented programming: A paradigm combining specification and implementation. In: Runtime Verification(RV'03), ENTCS, vol 89

Chen F, Roşu G (2007) MOP: An efficient and generic runtime verification framework. In: OOPSLA'07, ACM, pp 569–588

Chen F, Roşu G (2009) Parametric trace slicing and monitoring. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09), LNCS, vol 5505, pp 246–261

Chen F, D'Amorim M, Roşu G (2004) A formal monitoring-based framework for software development and analysis. In: ICFEM'04, LNCS, vol 3308, pp 357–372

Chen F, D'Amorim M, Roşu G (2006) Checking and correcting behaviors of Java programs at runtime with JavaMOP. In: Runtime Verification(RV'06), ENTCS, vol 144, pp 3–20

Chen F, Meredith P, Jin D, Rosu G (2009) Efficient formalism-independent monitoring of parametric properties. In: Automated Software Engineering (ASE'09), IEEE, pp 383–394

d'Amorim M, Havelund K (2005) Event-based runtime verification of Java programs. ACM SIGSOFT Software Engineering Notes 30(4):1–7

Drusinsky D (1997–2009) Temporal Rover. http://www.time-rover.com

Duncan AG (1978) Test grammars: A method for generating program test data. In: Workshop on Software Testing and Test Documentation, pp 270–281

Duncan AG, Hutchison JS (1981) Using attributed grammars to test designs and implementations. In: International Conference on Software Engineeing (ICSE'81), pp 170–178

Goldsmith S, O'Callahan R, Aiken A (2005) Relational queries over program traces. In: OOPSLA'05, ACM, pp 385–402

Hanford K (1970) Automatic generation of test cases. IBM Systems Journal 9(4):242–257

Havelund K, Roşu G (2001) Monitoring Java programs with Java PathExplorer. In: Runtime Verification(RV'01), ENTCS, vol 55

Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation, 2^{nd} edition. Addison-Wesley

Houssais B (1977) Verification of an algol 68 implementtion. In: Strathclyde Algol 68 Conference

Hughes G, Bultan T (2007) Interface grammars for modular software model checking. In: International Symposium on Software Testing and Analysis (ISSTA'07), ACM, pp 39–49

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: ECOOP'01, LNCS, vol 2072, pp 327–353

Kim M, Viswanathan M, Kannan S, Lee I, Sokolsky O (2004) Java-MaC: A run-time assurance approach for Java programs. Formal Methods in System Design 24(2):129–155

Knuth DE (1965) On the translation of languages from left to right. Information and Control 8(6):607–639

Leavens GT, Leino KRM, Poll E, Ruby C, Jacobs B (2000) JML: notations and tools supporting detailed design in Java. In: OOPSLA'00, ACM, pp 105–106

Martin M, Livshits VB, Lam MS (2005) Finding application errors and security flaws using PQL: a program query language. In: OOPSLA'07, ACM, pp 365–383

Maurer PM (1990) Generating test data with enhanced context-free grammars. IEEE Software 7(4):50–55

Meredith P, Jin D, Chen F, Roşu G (2008) Efficient monitoring of parametric context-free patterns. In: Automated Software Engineering (ASE '08), IEEE, pp 148–157

Purdom P (1972) A sentence generator for testing parsers. BIT 2:336–375

Schneider FB (2000) Enforceable security policies. ACM Transactions on Information System Security 3(1):30–50

Sirer E, Bershad B (1999) Using production grammars in software testing. In: Domain Specific Languagses (DSL'00), pp 1–13