Efficient Monitoring of Parametric Context-Free Patterns

Patrick O'Neil Meredith and Dongyun Jin and Feng Chen and Grigore Roşu
University of Illinois at Urbana-Champaign
{pmeredit, djin3, fengchen, grosu}@cs.uiuc.edu

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 of monitoring CFGs generate massive runtime overhead in real-life applications. This paper shows for the first time that monitoring parametric CFGs is practical *(on the order of 10% or lower for average cases, several times faster than the state-of-the-art). We present a monitor synthesis algorithm for CFGs based on an LR(1) parsing algorithm, modified with stack cloning to account for good prefix matching. In addition, a logic-independent mechanism is introduced to support partial matching, allowing patterns to be checked against fragments 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, like in traditional formal verification, but the code is checked against its requirements at runtime, like in testing. A large number of runtime verification approaches and systems, including JPax[19], TemporalRover[17], JavaMaC[22], Hawk/Eagle[16], Tracematches[4, 6], J-Lo[9], PQL[25], PTQL[18], MOP[14], etc., have been developed recently. In a runtime verification system, monitoring code is generated from the specified properties and integrated with the system to monitor. Therefore, a runtime verification approach consists of at least three interrelated aspects, namely (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, and is fundamental.

Monitoring safety properties can be arbitrarily complex[27], but recent developments in runtime verification show that regular and temporal logic formal specifications can be efficiently monitored against large programs. As shown by a series of experiments in the context of Tracematches[6] and JavaMOP[14], parametric regular and temporal logic formal specifications can be monitored against large programs with little runtime overhead: on the order of 10% or lower. However, both regular expressions and temporal logics reduce to ordinary automata when monitored, so they have an inherently limited expressivity: more specifically, most runtime verification approaches and systems consider only *flat execution traces*, namely, execution traces without any structure. Consequently, users of such RV systems are prevented from specifying and checking *structured properties*, referring to the structure of the program. 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". A more concrete example is given below.

1.1 Example

An important and desirable category of properties that cannot be expressed using regular patterns (with or without complement) 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

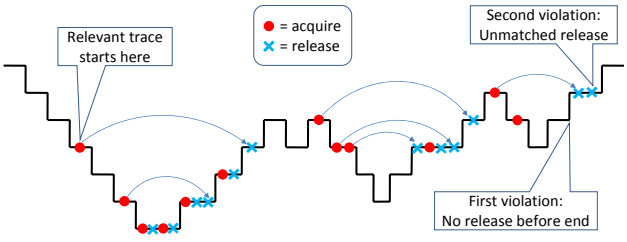

Figure 1. Example trace for structured acquire and release of locks

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, namely that 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 1.4+, 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 events acquire and release when the lock of interest is acquired and released, respectively, then here is a straightforward way to express this safety policy as a context-free grammar:

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

In other words, the pairs of events begin/end and acquire/release should match in a potentially nested way, respectively. S left-recursively refers to itself so that the pattern may be repeated. 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. A is necessary because the first event seen will be acquire[1] and we do not want to report violations for end's due to the lack of a matching begin.

It is clear that any (finished or unfinished) execution trace that is not a prefix of a word in the language of S in the CFG above is an execution that violates the safety policy. The CFG runtime verification technique presented in this paper and implemented as a logic-plugin in JavaMOP is able to monitor safety properties expressed as CFGs like above[2]. MOP and JavaMOP, the Java implementation of MOP, are discussed in Section 3.

Figure 2 shows this safe lock property expressed as a JavaMOP specification, using the CFG logic-plugin discussed in this paper. This specification makes use of JavaMOP's generic approach to parametric specifications described in[14]: specifications (and events) can be parametric, in this case in the lock `l`, and there could be one or more parameters. The logical formalisms in which properties are expressed need not be aware of the parameters (they need provide no special support for parameters); parameters are added automatically and generically by the JavaMOP framework. Only events which can begin a legal trace are monitor creation events, in other words, only the acquire event above can create new monitor instances. 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) code. We ran this property against a hand-crafted program, which generated the trace in Figure 1. Both violations were successfully caught because the CFG plugin throws away invalid events like a parser for a programming language, allowing it to catch multiple violations.

```
/*@
  scope = global
  logic = CFG
  SafeLock(Lock l) {
    event acquire<l> : begin(call(* l.acquire()));
    event release<l> : begin(call(* l.release()));
    event begin : begin(exec(* *.*(..)));
    event end : end(exec(* *.*(..)));
    start symbol : S;
    productions:
      S -> epsilon | S acquire M release A,
      M -> epsilon | M begin M end | M acquire M release,
      A -> epsilon | A begin | A end
    }
  Violation Handler{
    System.out.println("Unsafe lock operation found!");
  }
@*/
```

Figure 2. JavaMOP specification for the safe lock safety property using the CFG plugin

The interested reader can synthesize code for the specification in Figure 2 as well as any other, including all those discussed in the remainder of this paper, using JavaMOP's online interface[1](the above property is called SafeLock_CFG).

1.2 Contributions

Several approaches have been proposed to monitor context-free properties. For example, PQL[25] is based on context-free languages and Hawk/Eagle[16] uses a fix-point logic. These approaches propose rather complex solutions for monitoring CFG patterns and generate inefficient monitoring code in many cases, thus preventing the practical use of monitoring context-free properties. This paper shows that monitoring (the LR(1) subset of) parametric context-free patterns is *practical*. Our algorithm modifies an LR(1) parsing algorithm to generate monitors instead of parsers for the defined CFG and is totally different from the CFG monitoring algorithm in the PQL system[25]. We modify

[1]We wrote the CFG specification this way because we do not want the expense of monitoring begin and end for *all* methods.

[2]Our CFG plugin actually supports only the LR(1) language; when we use the term context-free we actually mean LR(1), unless explicitly mentioned otherwise.

the LR(1) parsing algorithm with a stack cloning process, which is needed to test for matches of prefix traces without altering the main monitor stack. In addition, generic partial trace matching is added to JavaMOP. Partial matching is defined as matching against every suffix of a given trace, and is the mode of matching used in PQL[25] and Tracematches[6]. We also describe optimizations that make partial matching more efficient.

An extensive evaluation of the CFG monitoring algorithm has been carried out, using the same benchmark (DaCapo) and some of the properties used previously for evaluating runtime verification systems[14, 10] (but now expressed as CFGs). Even when monitored using the CFG plugin, however, these regular pattern based specifications still use constant space. We thus perform 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 that monitoring context-free properties is useful. 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 that will be explained in Section 6. Another of the properties (SafeFileWriter), while expressible in PQL, is not expressible in Tracematches, because Tracematches only has limited ability to express structured properties, rather than the full generality of (deterministic) context-free languages.

Over both the adapted regular properties and the new strictly context-free properties, the overhead of JavaMOP with CFGs is, on average, over 6 times less than Tracematches on properties that Tracematches is able to express, and over 10 times less than PQL on properties that can be expressed in PQL. On all but 9 of the 66 benchmark/property pairs, the overhead is less than 5% in JavaMOP with CFGs.

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 partial matching together with its novel, optimized implementation in the context of JavaMOP. Section 5 explains our CFG monitor synthesis technique in JavaMOP, including considerations for partial 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

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to[14] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired and *no two of them* share the same logic. This observation strengthens our belief underlying MOP, that is, there probably is *no silver-bullet* specification formalism for all purposes. Also, most approaches focus on detecting either violations or validations (matches) 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. On the contrary, MOP was designed to be flexible for maximum extensibility and configurability.

Approach	Logic	Scope	Mode	Handler
JPax[19]	LTL	class	offline	violation
TemporalRover[17]	MiTL	class	inline	violation
JavaMaC[22]	PastLTL	class	outline	violation
Hawk[16]	Eagle	global	inline	violation
Tracematches[6]	Reg. Exp.	global	inline	validation
J-Lo[9]	LTL	global	inline	violation
PQL[25]	modified CFG	global	inline	validation
PTQL[18]	SQL	global	outline	validation

Table 1. Runtime Verification Breakdown

Specifically, there are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic answers which formalism is used to specify the property. The scope answers where to check the property; it can be class invariant, global, interface, etc. The running mode answers where the monitor 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)[3]. The handlers answer what to do if; 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. For example, for JavaMOP's extended regular expression plugin[14], one cannot use the validation of the pattern "$\sim$(a b)" ($\sim$ is the negation operator) to detect the violation of "a b", because, for example, an incomplete trace, "a", will cause a validation of "$\sim$(a b)" but should not cause the violation of "a b" since the monitor needs the next event to decide. Moreover, the negation of a context-free pattern may not even be context-free.

Most runtime monitoring approaches can be captured by instantiating the above attributes, as partly illustrated in Table 1. For example, JPax can be regarded as a monitor technique that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violations.

Of the systems mentioned in Table 1, only PQL[25] and Hawk/Eagle[16] can handle arbitrary context-free grammars. Hawk/Eagle adopts a fix-point logic and uses term rewriting during the monitoring, making it rather inefficient.

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

It also has problems with large programs because it does not garbage collect the objects it uses to monitor. In addition, Hawk/Eagle is not publicly available[4]. Because of this, we cannot compare our CFG plugin with Hawk/Eagle. In addition to PQL, we decided, also, to perform comparisons with Tracematches[6], 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[5], and also because it is a very efficient system.

3 MOP Revisited

MOP is an extensible runtime verification framework that provides efficient and 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 plugin, we have achieved an efficient runtime monitoring tool for universally quantified parametric CFG specifications.

In addition, we have implemented a novel extension of MOP, in JavaMOP, to support partial matching monitoring, also in a logic-independent way. We define partial 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. It is notable that PQL and Tracematches both support *only* partial matching, so in our experiments we use partial matching. In this section, we briefly introduce MOP.

3.1 MOP in a Nutshell

Monitoring-Oriented Programming (MOP)[13, 11, 14] is a formal framework for software development and analysis, in which the developer specifies desired properties using *definable* specification formalisms, along with code to execute when properties are violated or validated. Monitoring 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 plugins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture[13], 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 to use in the specification, one can also configure the behaviors of the generated monitor through different attributes[11]. 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 *generic* to the formalism used to specify the property. MOP also provides efficient and logic-independent support for universally quantified parameters in specifications[14], 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 plugin's implementation.

Unlike other approaches for parametric specifications, e.g., PQL[25] and Tracematches[6], MOP currently requires that all the parameters of a specification must be available in the monitor creation events. This deliberately accepted limitation allows for generating highly optimized monitoring code [14]. Although it limits the way of specifying properties, we have been able to write all the properties we considered (including those in [10] and [25]; see, as an example, the property in Figure 2) under this limitation. Moreover, our experience is that a carefully written specification can result in improved monitoring performance. Therefore, we have pragmatic reasons to believe that allowing "incompletely parameterized" events to create monitors is, perhaps, unnecessary. Nevertheless, the reader who disagrees with us can regard the CFG monitoring approach presented in this paper as one optimized for the common case: when the monitor creation events instantiate all parameters. It should also be noted that PQL allows for an unbounded number of parameters, while both JavaMOP and Tracematches are limited to a fixed number. We have not considered monitoring such properties using logic plugins in MOP, because (1) their handling is logic specific; and (2) MOP allows "raw" specifications to deal with such rare properties (see [14]).

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[12] to flexibly support these various interfaces, as well as for portability reasons. AspectJ[21] is employed for monitor integration: JavaMOP translates outputs of logic-plugins into AspectJ code, which is then merged within the original program by the AspectJ compiler. Five logic-plugins are currently provided with JavaMOP: Java Modeling Language (JML)[24], Extended Regular Expressions (ERE), Past-Time and Future-time Linear Temporal Logics (LTL) (see[12] for more details), and Context-Free Grammar

[4][6] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public[16]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from[10]. We have confirmed the inefficiency claims of[6] with the authors of Hawk/Eagle.

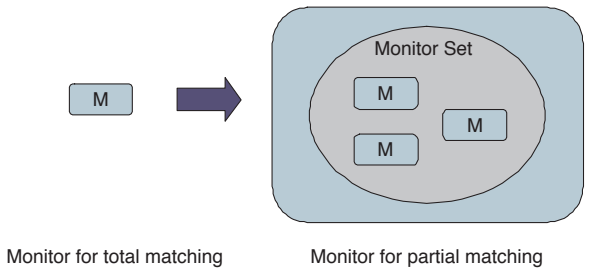

Figure 3. From Total to Partial Matching

(CFG) that is introduced in this paper. Note that these plugins 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 can yield very reasonable runtime overhead in practice, even for properties requiring intensive runtime checking: on the order of 10% or lower, and as efficient as the hand optimized monitoring code in most cases[14].

4 Partial Matching in JavaMOP

According to different requirements of applications, one may want to check the desired property against either *the whole execution trace* or *every suffix of a trace*. The former is called total matching, adopted by many runtime verification approaches to detect violations of properties, e.g., JPax[19] and JavaMac[22], and the latter is called partial matching, used mainly by monitoring approaches that aim to find matches of specifications, e.g., Tracematches[6] and PQL[25]. These two monitoring semantics can produce very different results for the same property. For example, for a regular pattern a^*b, a trace abb will trigger a validation at the first *b* and then a violation at the second *b* using total matching, while it will generate two matches at both *b*'s using partial matching. Partial matching usually only makes sense in conjunction with the validation of patterns, not the violation. Consider the pattern ab^* and the trace $abbbb$. This trace seems tailor made for the particular pattern, but every occurrence of *b* would generate a violation (even more confusingly, each occurrence would *also* generate a validation). Because of this we do not allow violation handlers in partial matching specifications.

The previous design of JavaMOP supported only total matching. One of our contributions here is that we have implemented a logic-independent extension of JavaMOP to support, also, partial matching. This extension is based on the observation that, although total matching and partial matching have inherently different semantics, it is not difficult to support partial 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*. 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 partial matching, as depicted by Figure 3. 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 operates as follows:

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

The third step is used to keep the "partial matching" monitor set small by removing unnecessary monitors.

Using our current implementation of partial matching in JavaMOP, one may further improve the monitoring efficiency if the monitor provides an optional interface, namely, 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 plugin, 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*.

Moreover, JavaMOP already requires the logic plugin to designate *creation events* that are the starting events of a validating trace in order to avoid unnecessary monitor creation. A new monitor instance needs to be created only at creation events. This feature is especially useful when combined with partial matching, which requires creating a new monitor at every event, if no creation events are chosen.

5 Context-Free Patterns in JavaMOP

We support the LR(1) subset of context-free grammars (CFGs). 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)[3, 20]. LR(1), however, is a large subset. Notably, LR(1) is more powerful than the standard mode of Bison[7], which provides LALR[5]. We base our implementation on the Knuth

[5] Bison also has a GLR mode, which can recognize all CFLs.

algorithm[23] for LR(1) parser table generation as presented in[3]. 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 more detail below.

5.1 Preliminaries

A CFG G, is formally defined as a tuple of the form, $G = (NT, \Sigma, P, S)$. Σ is the alphabet of the CFG, often referred to as the set of terminals. NT is the set of non-terminals. P is the set of productions, which define what strings non-terminals can derive. $NT \cup \Sigma$ is often called the set of *symbols* of the CFG. A production is always of 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 $\{a^n b^n | \Sigma = \{a, b\}\}$. As we can see, the non-terminal A represents the recursive nature of the language, as A can derive aAb an indeterminate amount of times before deriving ϵ.

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 is also used to decide which terminals in a 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, and *follow(A)*, may contain \$, the end of input token.

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-plugin Implementation

The CFG plugin 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. We have implemented the CFG logic plugin using the Maude term rewriting engine[15] and it is available for download at[1]. The operations of the plugin are described in the following sections.

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

Figure 4. CFG Monitoring Algorithm

5.3 CFG Simplification

The CFG plugin first simplifies the given grammar. The first step of simplification is the removal of non-generating non-terminals (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 non-terminals 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 has been removed from G, resulting in G', $L(G')$ = $L(G) - \epsilon$. This is acceptable, as a monitor matching the empty event trace has little utility.

5.4 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. 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. The tables are generated by an implementation of the full Knuth algorithm as presented in[3]. This algorithm is known to generate very large tables for complex grammars (such as those for programming language syntax). We believe that monitoring grammars will be relatively small, none-the-less we plan to implement the table compaction optimizations presented in[26] at a future date.

Pseudo-code for our monitoring algorithm is given in Figure 4. The significance of lines 14-17 is explained in the next section. An entry in *action_table* specifies, via the *action_type*, the type of action: *shift*, *reduce*, or *validate*. 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.

The *shift* action entry contains the next state for the DPDA in the *next_state* field. A *shift* action simply pushes the next state on the stack, if the next state is *not* the error state (lines 7-13). If, however, the table indicates that the next state *is* the error state, the algorithm reports a violation and breaks *without* touching the stack (lines 9-12). This allows the algorithm to continue to find more violations if total matching is in use. After a successful *shift* action, the while loop is broken, so that execution of the monitored program may continue until the next relevant event (line 18).

The *reduce* action is more complicated. The field *non_terminal* describes which non-terminal (A) the production $A \rightarrow \gamma$ reduces to, while the field *pop* denotes how many states to pop from the stack ($|\gamma|$). The reduction proceeds by popping the specified number of states from the stack and consulting *goto_table* to decide the next state (lines 19-25). 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 20). 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). If there were no *shift* operation for a given event, it would be effectively *ignored* (such as when it causes a violation).

The *validation* action, which directs the DPDA to validate the current input has no special fields, as no more information is necessary (lines 26-28).

5.5 Monitor Stack Cloning

Note that the 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 validation state can *only* be recognized when the next input is $. When the terminals are events generated dynamically, the end of the "string" (execution trace) cannot be known.

Our implementation of the algorithm attempts to reduce with $ as the look-ahead after *every* valid *shift* (lines 14-17). The problem with blindly reducing with $ as the look-ahead where possible is that all state of the current trace evaluation is lost. This means that the monitor can only validate the minimal trace that matches the CFG pattern. Since we desire to report validations for every sub-trace that matches the pattern[6], we must *clone* the stack before we perform any reductions with $ as the look-ahead.

This cloning ensures that the stack is intact for the next, and subsequent events, allowing for multiple validations. For example, consider the language denoted by the regular expression ab^*. While we would suggest using the ERE plugin for such a language, it is a clear example to illustrate the effect of cloning. With no cloning the algorithm would validate for only a. Because it popped during the validation phase, if it sees a b it will report failure. With cloning 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 cloning is to only clone 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 $\{a^n b^n | \Sigma = \{a, b\}\}$, only one clone is necessary no matter how long the input. Any grammar accepting unbounded repetition at the end of the pattern (like ab^*), will require cloning on each input of the repeated character.

5.6 Correctness of CFG Monitoring Algorithm

We next prove the correctness of our online monitoring algorithm for CFG. It is achieved by showing that our algorithm detects violations and validations of the observed trace in the same way as the Aho, Sethi, Ullman (ASU) parsing algorithm[3], as given in Figure 5.

Theorem 1. *For every* bounded *prefix of a (possibly infinite) trace and a CFG pattern, the MOP algorithm will notify a violation of the pattern if the ASU algorithm would notify a parse failure, and validation if ASU would notify success, given that prefix as total input.*

Proof. First, we construct a new parsing algorithm, as shown in Figure 6. This new algorithm can be easily proved to be equivalent to the one in Figure 5 as follows. The major difference between these two algorithm is that the pointer (*ip*) increment is moved to the outer loop in Figure 6. This change does not affect the behavior of the algorithm:

[6]This is irrespective of partial matching which actually generates multiple monitors.

```
1  globals stack, ip, at, gt
2  initialize stack.push(0), ip ← 0
3  procedure parse()
4    locals state, state', a
5    while (true) {
6      state ← stack.peek()
7      a ← get_token_at(ip)
8      switch (at[state, a].action_type) {
9      case shift :
10       state' ← at[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(at[state, a].pop)
21       state' ← stack.peek()
22       stack.push(gt[state', at[state, a].non_nonterminal])
23       continue while
24     case accept :
25       accept
26       return
27     }
28   }
```

Figure 5. Aho, Sethi, Ullman Algorithm

```
1  globals stack, ip, at, gt
2  initialize stack.push(0), ip ← 0
3  procedure parse()
4    locals state, state', a
5    while (true) {
6      a ← get_token_at(ip)
7      while (true) {
8        state ← stack.peek()
9        switch (at[state, a].action_type) {
10       case shift :
11         state' ← at[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(at[state, a].pop)
20         state' ← stack.peek()
21         stack.push(gt[state', at[state, a].non_nonterminal])
22         continue while
23       case accept :
24         accept
25         return
26       }
27     }
28     advance ip
29   }
```

Figure 6. Modified ASU Algorithm

1. For a shift action, both algorithms carry out the same operation except that Figure 5 increases the pointer and continues to the next action, while Figure 6 breaks the inner loop, increases the pointer in the outer loop, and then continue to the next action. Obviously, both are equivalent.

2. For reduction, Figure 6 chooses to stay in the inner loop, which is identical to Figure 5, which continues the loop without increasing the point.

3. For acceptance, both algorithms are identical.

Now we can prove the correctness of the monitoring algorithm in Figure 4 by comparing it with the modified parsing algorithm in Figure 6.

The major difference distinguishing the monitoring algorithm from the parsing algorithm is that the former has to wait for the next event extracted from the execution of the monitored program while the latter can actively retrieve the next token, which is handled in the outer loop in Figure 6. Therefore, we only need to prove that the *monitor* procedure in Figure 4 produces the same result as the inner loop in Figure 6, given the same state and event to process.

It is straightforward to compare both pieces of code and the only difference between them is the stack cloning (line 14 to 17) in Figure 4. It is needed because we wish to continue parsing *after an accept*, and because we can never actually see $ as an event (an online monitor does not see the end of the execution). we clone 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 on this recursive call, because $ can never be shifted[7]. Due to this, the recursion is 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 once and only once for each symbol seen (that does not cause an error). Because the algorithm repeats until a shift action, error, or accept happens, we ensure that the recursive call must happen, if it happens, after the processing of each input[8]. Cloning the stack allows us to reduce and accept, while still maintaining the original stack to continue parsing events as if the end of input had *not* 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 failure if it would report failure, and that it additionally retains enough state to continue parsing future prefixes.

[7] This is a property of the table generation algorithm, which we use without proof. We feel it is intuitive, however.

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

□

5.7 Partial Matching with CFGs

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

The first is identification of monitor creation events[9]. 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 plugin are those events which are in *first(S)*, where S is the start symbol for the grammar.

Additionally, it is necessary to define a hash encoding for CFG based monitors because our partial 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). Lastly, we need an equality method 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 CFG monitors to be equivalent, as they do not have finite state like the other logic plugins 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.

6 Evaluation

We evaluated the CFG plugin on the Dacapo benchmark suite[8]. Also, we evaluated PQL and Tracematches against DaCapo using the same properties for comparison.

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[8]: `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 last six iterations is then used to compute the runtime overhead. Therefore, Table 2 percentages should be read "±3".

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

6.2 Properties

The following general properties borrowed from[10] 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 plugin; they are all properly context-free:

- ImprovedLeakingSync: The original LeakingSync specified in[10] *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 choose the first three regular-language-based properties (HashMap, HasNext, and SafeIterator) to include in this paper because they generate a comparatively larger runtime overhead, and because they are all expressible in PQL[10]. 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 partial matching mode; the decentralized indexing of monitors was used in all the experiments[14]. We chose the AspectJ compiler 1.5.3 (AJC) in the evaluation to compile the JavaMOP generated monitoring AspectJ code. For Tracematches we used the most recent published version from[2]. All of these properties are available on the JavaMOP Online Interface[1]. Each one is named Partial<PropertyName>_CFG.

6.3 Results

Table 2 shows the percent overheads of JavaMOP using the CFG plugin, PQL, and Tracematches. N/E refers to specifications that were not expressible. Tracematches is

[10]Some of the other regular language properties we have evaluated require arbitrary side effects. PQL has no faculties for this.

	HashMap			HasNext			SafeIterator			ImprovedLeakingSync			SafeFileInputStream			SafeFileWriter		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	3	6	0	1	2	3	2	82	0	1	N/E	N/E	3	113	-1	2	22	N/E
bloat	14	9	-2	1112	5929	2452	627	8694	11258	13	N/E	N/E	1	128	0	27	97	N/E
chart	-1	1	-1	-1	3	0	2	50	11	4	N/E	N/E	0	29	1	0	37	N/E
eclipse	0	1	1	0	2	-1	-2	1	2	1	N/E	N/E	-2.5	3	0	-2	1	N/E
fop	3	2	0	0	2	-1	-1	24	5	1	N/E	N/E	-2	58	-1	-2	47	N/E
hsqldb	0	3	15	0	6	15	0	78	17	1	N/E	N/E	1	280	21	2	95	N/E
jython	0	23	15	0	0	13	0	12	16	41	N/E	N/E	0	937	12	1	crashes	N/E
luindex	1	8	1	-2	93	2	3	181	9	1	N/E	N/E	-1	233	6	0	33	N/E
lusearch	1	1	8	-1	59	9	4	132	34	2	N/E	N/E	-1	137	7	0	49	N/E
pmd	-1	0	3	191	1870	52	178	1334	175	36	N/E	N/E	-1	547	1	-2	658	N/E
xalan	0	5	1	0	0	2	1	53	10	3	N/E	N/E	-1	90	3	-2.5	164	N/E

Table 2. Average percent runtime overhead for JavaMOP CFG (MOP), PQL, and Tracematches (TM) (average of 6 trials after convergence within 3%)

Property	HashMap	HasNext	SafeIterator	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
antlr	0	0	1990	8472	0	385
bloat	361519	146628987	75944328	5587905	259	0
chart	8773	0	569345	634260	0	0
eclipse	20888	0	32759	74630	930	0
fop	17265	0	49959	182407	12	0
hsqldb	0	0	0	0	0	0
jython	443	0	177554	23969673	544	0
luindex	9615	0	82162	1559386	1114	0
lusearch	416	0	405428	1291992	0	32
pmd	11354	36163717	25476563	26291289	10	0
xalan	124155	0	1009649	5146036	13604	0

Table 3. Number of events handled by JavaMOP with CFGs

Property	HashMap	HasNext	SafeIterator	ImprovedLeakingSync	SafeFileInputStream (**)	SafeFileWriter
antlr	0	0	0	0	0	385
bloat	361436	73232724	1894581	0	3	0
chart	8729	0	815	0	0	0
eclipse	15836	0	588	0	8	0
fop	16980	0	79	0	0	0
hsqldb	0	0	0	0	0	0
jython	443	0	50	0	48	0
luindex	9615	0	8788	0	0	0
lusearch	416	0	0	0	0	32
pmd	116	10093591	1950212	0	2	0
xalan	95252	0	0	0	0	0

Table 4. Number of instantiated total match monitors

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

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 running of the entire DaCapo benchmark suite, over 355 *million* events (Table 3) are generated, creating more than 87 *million* monitors (Table 4). Notably, the highest runtime overheads are associated with the larger number of monitors, more-so than the number of events (for example, ImprovedLeakingSync monitors a large amount of events but generates no monitors, so the overhead remains fairly low). This is mainly due to the amount of time spent in indexing and attempting to merge monitors.

The average overhead of JavaMOP over the 66 program/property pairs is 34%. There are two considerations here, however: (1) we chose specifically those properties that generated the largest overheads, (2) when the two largest overheads are removed, the average over the remaining 64 pairs drops to a very reasonable 8%. Further, the average JavaMOP overhead for properties expressible in PQL was 39% over 55 pairs, while PQL's overhead on these same properties was 415%. Similarly, for Tracematches expressible properties JavaMOP's, overhead was 48% over 44 pairs, while Tracematches was 322%. Tracematches, PQL, and JavaMOP all feature the same two pairs which have extremely large overhead compared to the median (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 8%, while PQL still weighs in at 150%. Tracematches is comparable to JavaMOP, with JavaMOP at 9% and Tracematches at 11%. 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 6 times lower.

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 partial matching than it does matching the pattern, while PQL and Tracematches overheads are more affected by the complexity of the pattern. Table 4 supports this claim, showing the HasNext generates far more monitors than SafeFileInputStream. Note that JavaMOP with CFGs far outperforms both PQL and Tracematches on these 2 program/property pairs.

SafeFileInputStream is an interesting case: as a context-free grammar 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 may generate relevant events.

The memory overhead of monitoring 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 about the memory overhead caused by JavaMOP monitors, namely, bloat-SafeIterator and pmd-SafeIterator. Our investigation shows that both programs, bloat and pmd, make intensive use of some vectors and create numerous iterators to do computation over the vectors throughout the whole 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, because for HasNext, one monitor is 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 and much lower memory overhead was caused. Compared with the results of Tracehmatches, we believe there is still some room for improvement with regard to memory usage in our approach. We are working on a new technique that is based on the idea of removing "'dead"' monitors using static analysis on the specification.

7 Conclusions and Future Work

We implemented a CFG pattern based logic plugin for JavaMOP using a modified LR(1) parsing algorithm. Our modification to the algorithm is based on the novel idea of *cloning* the stack in order to "predict" a possible reduction with $ (end of string) as a look ahead without destroying the state of the monitor. 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 partial trace matching in order to fairly compare JavaMOP with PQL, which uses a proprietary logic encompassing context-free languages, and Tracematches. Tracematches, however, cannot handle arbitrary CFG patterns.

We have also begun work on a logic called PtCaRet as another specification formalism to support structured specifications. PtCaRet is past time temporal logic with calls and returns. We plan to implement both static and dynamic soon. Further, we intend to add the Pager[26] optimizations to the CFG plugin table creation algorithm in order to reduce the size of the created action and goto tables.

References

[1] JavaMOP CFG website links. Java MOP CFG Plugin: `http://fsl.cs.uiuc.edu/index.php/JavaMOPCFG`, JavaMOP Online Interface: `http://fsl.cs.uiuc.edu/index.php/Special:JavaMOPOnline`.

[2] Tracematches Benchmarks. `http://abc.comlab.ox.ac.uk/tmahead`.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986. pages 215-246.

[4] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, 2005.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *AOSD*, 2005.

[6] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA*, 2007.

[7] Bison. `http://www.gnu.org/software/bison/`.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[9] E. Bodden. J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

[10] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.

[11] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *ICFEM*, 2004.

[12] F. Chen, M. D'Amorim, and G. Roşu. Checking and correcting behaviors of Java programs at runtime with JavaMOP. In *Runtime Verification*, 2005.

[13] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification*, 2003.

[14] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer-Verlag New York, Inc., 2007.

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

[17] Temporal Rover. `http://www.time-rover.com`.

[18] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, 2005.

[19] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification*, 2001.

[20] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. Addison-Wesley, 2001.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[22] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. JavaMaC: a runtime assurance tool for Java. In *Runtime Verification*, 2001.

[23] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec. 1965.

[24] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA*, 2000.

[25] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[26] D. Pager. A practical general method for constructing LR(k) parsers. *Acta Information*, 7:249–268, 1977.

[27] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security*, 3(1):30–50, 2000.