Efficient Parametric Runtime Verification with Deterministic String Rewriting *

Patrick Meredith
University of Illinois at Urbana-Champaign
Urbana IL, USA
pmeredit@illinois.edu

Grigore Roşu
University of Illinois at Urbana-Champaign
Urbana IL, USA
grosu@illinois.edu

ABSTRACT

Early efforts in runtime verification and monitoring show that parametric regular and temporal logic specifications can be monitored efficiently. These approaches, however, have limited expressiveness, since their specifications always reduce to monitors which are finite state machines. More recent developments showed that parametric context-free properties can be efficiently monitored with overheads generally lower than 12–15%. While context-free grammars are more expressive than finite-state approaches, they still do not allow every computable safety property. This paper presents a monitor synthesis algorithm for string rewriting systems (SRS). SRSs are well known to be Turing complete, allowing for the formal specification of any computable safety property. Earlier attempts at Turing complete monitoring have been relatively inefficient. This paper demonstrates that monitoring parametric SRSs is practical. The presented monitoring algorithm uses a modified version of the Aho-Corasick string searching algorithm for quick pattern matching with an incremental rewriting approach that avoids reexamining parts of the string known to contain no redexes.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion Checkers, Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

General Terms

Algorithms, Reliability, Security, Verification

Keywords

Runtime Verification, Monitoring, String Rewriting

*Supported in part by NSF grant CCF-0916893 and by NSA contract H98230-10-C-0294.

1. INTRODUCTION

Runtime verification (RV) is a 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 techniques and systems, including TemporalRover [18], JPaX [20], JavaMaC [23], Hawk/Eagle [17], Tracematches [2, 3], J-Lo [8], PQL [25], PTQL [19], MOP [15, 14], Pal [12], RuleR [5], etc., have been developed recently, and the overall approach has gained enough traction to spawn its own conference [4]. 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: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a means to instrument programs. The chosen specification formalism determines the expressivity of the runtime verification approach and/or system.

Monitoring safety properties is arbitrarily complex [28]. Early developments in runtime verification, showed that *parametric* regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. A parametric monitor associates monitor states with different object instantiations for the given parameters. This allows for specification of properties about the relationships of objects, e.g., a relationship between a Collection object and its associated Iterator objects in Java[1]. As shown by experiments with Tracematches [3] and the most recent experiments using JavaMOP [21], parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 15% or lower.

However, both regular expressions and temporal logics are monitored using finite automata, 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. PQL [25], Hawk/Eagle [17], and RuleR [5] provide more expressive logics, but these are relatively inefficient [2, 3, 15]. More recently, JavaMOP was extended to support efficient context-free monitors with runtime overheads very similar to

[1]Typestates [29], a popular concept in software engineering and software analysis, can be monitored with parametric monitors that have only one parameter.

```
public class RandomEquality {
  int numberOfNumbers;

  public RandomEquality(int numberOfNumbers){
    this.numberOfNumbers = numberOfNumbers;
  }

  public int nextNumber() {
    return genNextNumber(numberOfNumbers--);
  }

  public boolean hasNextNumber(){
    return numberOfNumbers > 0;
  }

  private int genNextNumber(int currentNumber){
    //some logic that may or may not be correct
  }
}
```

Figure 1: A Java class that provides random number sequences of any length that maintain equality

the earlier finite-state logics [26]. While this work allows for checking many structured properties, it does not have the full power to specify any possible safety property. In this paper, we introduce an algorithm for monitoring parametric deterministic string rewriting systems, to serve as an efficient runtime verification technique for specifying and monitoring arbitrarily complex properties; indeed, string rewrite systems are known to be as expressive as Turing machines [11]. We also provide an implementation of our algorithm as an MOP *logic plugin* [15, 14], so it can be used as integral part of the JavaMOP runtime verification system. By abuse of vocabulary, we will refer to deterministic string rewriting systems as string rewriting systems and abbreviate them SRSs.

1.1 Examples

Safety properties that require more expressivity than a context-free language are generally more intimately related to the specifics of the program under verification/test than those that may be monitored using context-free or finite logics. Conversely, less specific properties, such as correct API usage, tend to be finite state [24]. As a relatively simple, and admittedly contrived, example of a non context-free property, consider the Java class, RandomEquality defined in Figure 1. The idea of this class is to provide a random string of numbers from the set $\{0, 1, 2\}$ of a given length defined by the parameter passed to the constructor that maintains equality, that is, that the number of 2's is equal to the number of 1's is equal to the number of 0's.

The JavaMOP specification presented in Figure 2, which uses the new `srs` logic plugin, is able to catch any failures of this class to provide equality. JavaMOP specifications begin with a declaration of the name of the specification and parameters. Here the property is named EqualityCheck, and one parameter re of type RandomEquality. The parameters allow us to associate separate monitor states with each object instantiation of the parameters. In this case, with one parameter, there will be one monitor state associated with each object instance of RandomEquality in the program under test. This is important because we would not want

```
EqualityCheck(RandomEquality re) {
  event done after(RandomEquality re)
            returning(boolean b) :
          call(* RandomEquality.hasNextNumber())
          && target(re) && condition(!b) {}
  event e0  after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 0) {}
  event e1  after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 1) {}
  event e2 after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 2) {}
  srs :
   e1 e0 -> e0 e1 . e2 e0 -> e0 e2 . e2 e1 -> e1 e2 .
   e0 e1 -> E        .
   E e1  -> e1 E     . E e0  -> e0 E     .
   E e2  -> #epsilon . e2 E  -> #epsilon .
   ^  done   -> #succeed .
   e0 done   -> #fail    .
   e1 done   -> #fail    .
   e2 done   -> #fail    .

  @succeed {
    System.out.println(
      p.toString() + " worked perfectly!"); }
  @fail {
    System.out.println(
      p.toString() + " failed!"); }
}
```

Figure 2: A JavaMOP specification that finds equality failures in the RandomEquality class

calls to different object instances of the RandomEquality class to interfere with each other as such would assuredly lead to false positives and negatives.

The next part of a JavaMOP specification is the declaration of events. Here we are able to generate four different events: done, e0, e1, and e2. The events are defined using a superset [27] of AspectJ [22] advice with embedded pointcuts. Here, the event done is defined to occur when the hasNextNumber() method is called and returns false, signalling the end of the randomly generated number string[2]. The events e0, e1, and e2 all correspond to calls of nextNumber() where the proper number in $\{0, 1, 2\}$ is returned.

After the event definitions, we list the formalized property. The keyword srs tells JavaMOP that the following property will be a deterministic string rewriting system. Rules in our SRS formalism take the form "l → r .", meaning that the string of events on the left hand side of the arrow rewrites to that on the right side. The three rules on the first line of this SRS sort the events: all e0 come before all e1 which come before all e2. The rule e0 e1 → E denotes that we have

[2]Note that this requires properly calling hasNextNumber() before calling hasNextNumber(). This can be ensured in JavaMOP using a different, finite state, property.

found a pair of e0 and e1, which must be matched by an e2.

Note that the SRS rules can be applied in any order when a new event is received, so it is user's responsibility to write *confluent* SRSs or to use the deterministic order of rule application explained in Section 3.1. The two rules on the third line move all instances of E to the right, so that they will eventually become adjacent to any instances of e2. The two rules on line four correspond to when such a situation occurs. When E is adjacent to e2 we have found a triple, and we can safely remove the symbol E from the string by rewriting it to #epsilon, which is a keyword specific to the SRS formalism in JavaMOP. The next rule is the success case, which occurs when done is at the beginning of the string (denoting that all 0's, 1's, and 2's have been equal). The symbol ^ corresponds to the beginning of the string. Similarly, $ corresponds to the end, but is not used here. #succeed is a special keyword that stops the rewriting process with the monitor signalling that a success was found. Like #succeed, #fail is a keyword that stops the monitor with a failure returned.

The next three lines are the failure cases; each rewrites to #fail. They occur when the number of 0's, 1's, and 2's was not equal, because the string will always be empty when a done occurs if they were properly balanced. The failure cases rely on the incremental nature of the string rewriting process. If taken as a normal SRS with complete strings as input, this would not be confluent. That means, the choice of what order to apply rules would result in different normal forms (See Section 3.1). Because the normal form is computed between the arrival of each event, e0, e1, or e2 can only occur before done in the string if an unequal number occurred. The string rewriting process is explained fully in Sections 3.2–3.4.

The last part of a JavaMOP specification is the handler section. Handlers are arbitrary Java code that is executed when the monitor raises a particular condition. Here the keywords @succeed and @fail denote that the code within the subsequent braces is run when the string rewrites to #succeed or #fail, respectively. In this example, the handlers simply print out informative messages when such situations occur. In general, handler code may be used for anything, such as running a specific algorithm or recovering from the error denoted by the failure of the safety property in question.

Aside from specifying properties which cannot be expressed by context-free grammars, string rewriting systems can be useful for expressing context-free and finite properties in a natural, and often times more compact, form. Below are two properties from earlier papers [26, 27] written as SRSs.

The first property, called HasNext, is a property of the Java Iterator interface stating that hasNext() should always be called and return true before next is called. Below it is specified as a regular expression:

$$(\text{hasnexttrue next})^* \text{ next}$$

The corresponding SRS is as follows:

$$\begin{aligned} \text{hasnexttrue next} &\rightarrow \text{\#epsilon} \\ \text{hasnexttrue hasnexttrue} &\rightarrow \text{hasnexttrue} \\ \text{\^{}next} &\rightarrow \text{\#fail} \end{aligned}$$

While this SRS is certainly larger than the original ERE, it may be easier to understand by some users because it directly captures the semantics of the property by simply enumerating all the cases that one has to worry about. The rule hasnexttrue hasnexttrue → hasnexttrue conveys the notion that multiple calls to the hasNext() method are idempotent. hasnexttrue next rewrites to #epsilon because it is a safe operation. If next is seen at the beginning of the string a failure is raised as hasnexttrue was not properly called. Because our algorithm is incremental and deterministically rewrites from left to right it is not strictly necessary to match the beginning of the string, but it is more clear conceptually.

The second property is a properly context-free property called SafeLock which corresponds to the proper nesting of acquiring and releasing locks. Proper nesting, in this case, means that corresponding calls to acquire() and release() occur within the same method body. Here begin and end denote the beginning and end of a method body.

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

The property is fairly complex, and a complete explanation can be found in [26]. The SRS for the property follows:

$$\begin{aligned} \text{begin end} &\rightarrow \text{\#epsilon} \\ \text{acquire release} &\rightarrow \text{\#epsilon} \\ \text{begin release} &\rightarrow \text{\#fail} \\ \text{acquire end} &\rightarrow \text{\#fail} \end{aligned}$$

In this case, the SRS is quite a bit less complex than the context-free grammar specifying the same safety property. Again, it conveys interesting semantic information. From the SRS it is clear that a begin followed immediately by a release() results in an error because we require all release() to occur in the same method call as the corresponding acquire(). Similarly, an acquire() follow by a end results in an error because the lock is not correctly released within the method body. begin end and acquire release rewrite to #epsilon because they are properly nested when they occur adjacently.

1.2 Contributions

There are two main contributions to this paper:

- An efficient, optimized string rewriting algorithm. It builds upon a modification of the Aho-Corasick algorithhm [1]. The original algorithm was designed for quickly finding strings in text. Our modified algorithm keeps track of substitution boundaries so that a rewrite step can be performed in time linear to the length of the right hand side of the matched rule[3]. To our knowledge, this is the first time it has been applied to string rewriting. An optimization has also been devised, which checks for early termination of rewriting.
- An implementation and extensive evaluation of the above algorithm as an MOP logic plugin for runtime verification. This way, it can serve as a specification formalism for parametric safety properties in instances of the MOP framework, such as JavaMOP. We show that its performance in practical runtime verification of large systems is acceptable when compared to other means to specify the same properties. Additionally, we show that it outperforms one of the state-of-the-art rewrite engines, Maude [16], which implicitly supports string rewriting as rewriting modulo associativity.

1.3 Paper Outline

Section 2 presents related work in the field of runtime verification: popular runtime monitoring systems, with a

[3]The right hand side must be copied, so that the rule is still viable the next time it matches.

Approach	Logic	Scope	Mode	Handler
JPaX [20]	LTL	class	offline	violation
TemporalRover [18]	MiTL	class	inline	violation
JavaMaC [23]	PastLTL	class	outline	violation
Hawk [17]	Eagle	global	inline	violation
RuleR [5]	RuleR	global	inline	violation
Tracematches [3]	Reg. Exp.	global	inline	validation
J-Lo [8]	LTL	global	inline	violation
Pal [12]	modified Blast	global	inline	validation
PQL [25]	PQL	global	inline	validation
PTQL [19]	SQL	global	outline	validation

Figure 3: A Selection of Monitoring Systems

particular emphasis on those with greater than finite state specification languages and on our framework of choice for our implementation and experimental testbed, MOP. Section 3 presents our string rewriting algorithm, with its use and construction of pattern match automata and and optimization that allows for early termination. Section 4 presents our experimental results, and Section 5 concludes.

2. RELATED WORK AND MOP

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to [27] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired, and few of them share the same logic.

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). The handlers specify what actions to perform under exceptional conditions; such conditions include violation and/or validation of the property. It is worth noting that for some 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. MOP allows for handlers for any number of user defined exceptional situations (called handler categories).

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Figure 3, which shows an (incomplete) summary of runtime monitoring systems. 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 (the Java instance of MOP) has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Of the systems mentioned in Figure 3, only PQL [25], Hawk/Eagle [17], and RuleR [5] provide logical formalisms with greater than finite-state power. Hawk/Eagle adopts a Turing-complete fix-point logic, but it has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available[4]. Because of this and the fact that Hawk/Eagle has not been run on DaCapo [7] with the same properties, we cannot compare JavaMOP with our new string rewriting systems plugin with Hawk/Eagle. RulerR is a rule-based monitoring system which has the ability to also specify Turing complete properties. The current implementation of RuleR is not built for efficiency, and is, additionally, not publicly available. PQL is not Turing-complete, and performance comparisons with PQL using an older, less efficient, version of JavaMOP can be found in [26]. String rewriting was used in the context of monitoring for detection of malware in [6]. This was, in many ways, the inspiration for adding string rewriting to MOP. However, the string rewriting patterns allowed in that work were regular (i.e., can capture only regular languages), while our goal is to provide a true Turing-complete logical formalism for parametric monitoring.

MOP [15, 14] is an extensible runtime verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an instance of MOP for the Java programming language. It allows the developer to specify desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. 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 plug-ins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture [14], which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. MOP also provides efficient and logic-independent support for *parametric* parameters [13], which is useful for specifying properties related to groups of objects. 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 parametric patterns simplified our SRS plug-in's implementation.

The JavaMOP instance provides two interfaces: a web-based interface and a command-line interface, providing the developer with different means to manage and process JavaMOP specifications. AspectJ [22] is employed for monitor integration: JavaMOP translates outputs of logic plug-ins into AspectJ code, which is then merged within the original program by an AspectJ compiler. Seven logic-plug-ins are currently provided with JavaMOP: finite state machines, extended regular expressions, context-free grammars, past time linear temporal logic, linear temporal logic with past and future operators, past time linear temporal logic with calls and returns, and, now, string rewriting systems. Descriptions of the first six plugin-ins can be found in [27].

3. MONITORING SRS SPECIFICATIONS

In this section, we present some basic notation for string rewriting systems and our string rewriting algorithm which was implemented as a logic plugin in the MOP framework.

[4][3] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [17]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [10]. We have confirmed the inefficiency claims of [3] with the authors of Hawk/Eagle.

3.1 Preliminaries

We refer the reader to [11] for an in-depth presentation of string rewrite systems. For an alphabet Σ, a *string rewriting system* (SRS) is a binary relation, R, on Σ, that is, a subset of $\Sigma^* \times \Sigma^*$. The set $\{\mathsf{l} \in \Sigma^* \mid (\mathsf{l}, \mathsf{r}) \in R\}$ is called the *domain* of R, denoted $dom(R)$, while similarly the set $\{\mathsf{r} \in \Sigma^* \mid (\mathsf{l}, \mathsf{r}) \in R\}$ is called the *range*, denoted $range(R)$. We refer here to any element $(\mathsf{l}, \mathsf{r}) \in R$ as a *rule* in R, any $\mathsf{l} \in dom(R)$ as a *left hand side (LHS)* of a rule in R, and any $\mathsf{r} \in range(R)$ as a *right hand side (RHS)* of a rule in R. In our SRS specifications in this paper and in JavaMOP, rules $(\mathsf{l}, \mathsf{r}) \in R$ are written using the earlier shown syntax "$\mathsf{l} \rightarrow \mathsf{r}$".

The *single-step reduction relation* on Σ^* that is induced by R is defined as: for any $u, v \in \Sigma^*$, $u \rightarrow_R v$ if and only if there exists $(\mathsf{l}, \mathsf{r}) \in R$ such that for some $x, y \in \Sigma^*$, $u = x\ \mathsf{l}\ y$ and $v = x\ \mathsf{r}\ y$. The *reduction relation* on Σ^* induced by R is the reflexive, transitive closure of $\rightarrow_R$ and is denoted by $\rightarrow_R^*$. If for $x, y \in \Sigma^*$, $x \rightarrow_R^* y$ and y is irreducible, y is a *normal form* for x. R is *confluent* if there is only one such y for any given x, regardless of the order in which rules are applied.

In our SRSs in MOP, the symbols $s \in \Sigma$ correspond to either *events* of our property or symbols that appear in the RHS of rules in R. We call our string rewriting systems *deterministic* because the same normal form will always be chosen in the presence of a non-confluent R. Specifically, rules are applied left-to-right, with the smallest rule matching first in the case of overlap (e.g., for LHSs a a and a a b, the rule with a a as its LHS will always be applied first, starving the other rule). In the case of a conflict that is not resolved by the above, the order of rules in the SRS specification is used to determine which rule to apply (e.g., if two rules have the same LHS, the one specified first will always be applied).

3.2 String Rewriting Algorithm Overview

There are two major parts to our SRS algorithm:

1. Finding matches of the LHSs of rules; and
2. Performing replacements with RHSs of rules.

To make replacements as efficiently as possible, the string of events/symbols that we rewrite is a linked listed of the SpliceList class, which was specially created for our purposes to allow constant time replacement of a section of the list with another list (splicing). The SpliceList class has a special type of Iterator defined for it, called the SLIterator, that does *not* follow the normal Iterator interface in Java.

Rather than only having next() and hasNext() methods, the SLIterator has next(int i), which moves the SLIterator forward i times and returns true if it is successful (i.e., does not reach the end of the SpliceList), and get(), which returns the current element that the SLIterator points to. SLIterator also has a method, splice(SLIterator second, SpliceList replacement), which takes another SLIterator to the same SpliceList and replaces the sequence denoted by those two SLIterators, inclusively, by a specified sequence replacement. It is because of the inclusive nature of the splice method that the SLIterator must have a method to retrieve its current element without advancing. The splice method makes it imperative for our string matching algorithm to maintain SLIterators to the beginning and end of the current LHS under consideration.

In Section 3.3 we discuss how this matching occurs using a modification of the Aho-Corasick string searching algorithm [1] that, unlike the base algorithm, keeps track of the beginning of a match, so that rewrites can be performed in constant time (after copying the RHS in time proportional to its length). To make the paper self-contained, we give all the necessary information regarding the Aho-Corasick algorithm, rather than only this modification, but the modification is clearly delineated. To our knowledge, this is the first time any variation on the Aho-Corasick algorithm has been used in string rewriting, and no implementations of SRSs exist, that we could find. In Section 3.4, we present an in-depth explanation of how the pattern matching fits into the string rewriting algorithm and how we optimize string rewriting to avoid considering sequences that cannot match any LHS.

3.3 Pattern Match Automata

The pattern match automata used by our string rewriting process, as mentioned, is a modification of the Aho-Corasick algorithm for finding strings in text [1]. The Aho-Corasick algorithm, which was originally not designed for string rewriting, is able to find all matches in a string in one linear pass, rather than performing separate passes for each rule LHS as would a naive matching algorithm. Our modification of the algorithm allows us to correctly adjust the SLIterator to the beginning of our current match, facilitating quick rewrites.

3.3.1 Using Pattern Match Automata

Figure 4 shows the pattern match automaton for the SafeLock property. Each node has at least its state number and state depth, listed as a pair *number:depth*. The depth is used in two places in the automata generation algorithm, and simply states how many symbols (events) have been processed since the start state in one of the LHSs of the rewrite rules in our SRS. This will be explained in more detail below. Additionally, states which correspond to matching the left hand side of a given rule also display that rule, e.g., in state 6, the begin release $\rightarrow$ #fail rule is matched. Each edge is marked by the list of symbols that cause that transition, as well as a number following a "/". That number, which we refer to as the *action*, is the number of times to increment the *first* SLIterator except in the self-transitions of state 0. When a self-transition in state 0 occurs, the *first* Iterator must be incremented once. When a forward transition with "/ 0" is encountered, a transition to the next state is made, and the next input is considered. If the transition is suffixed with something *other* than 0, the transition must be a backward transition, and the same symbol that is currently under consideration must be evaluated in the next state. This is why we handle self-transitions in state 0 as a special case, if it were suffixed with "/ 1" and handled as a backward transition, the same symbol would be considered infinitely.

Figure 5 shows the pseudocode for pattern matching using a given pattern match automaton. The only global variable for the algorithm is the given PatternMatchAutomaton, *pma*. The algorithm begins by initializing the *first* and *second* SLIterators to the beginning of the argument SpliceList l, using the *head()* method. The local *currentState* is initialized to the initial machine state, here represented as 0[5]. The while loop beginning on line 10 will only exit when the end of l is reached, denoted by the break statements on lines 20 and 25. We know that the end of l is reached on lines 20 and 25 when the next(int i) method returns false. We never need to check if *first.next* returns false because it may never advance past *second* due to the construction of the PatternMatchAutomaton. Lines 17–22 cover the self transition

[5]It is actually a class that may contain a matched rule, as we can see in Figure 4.

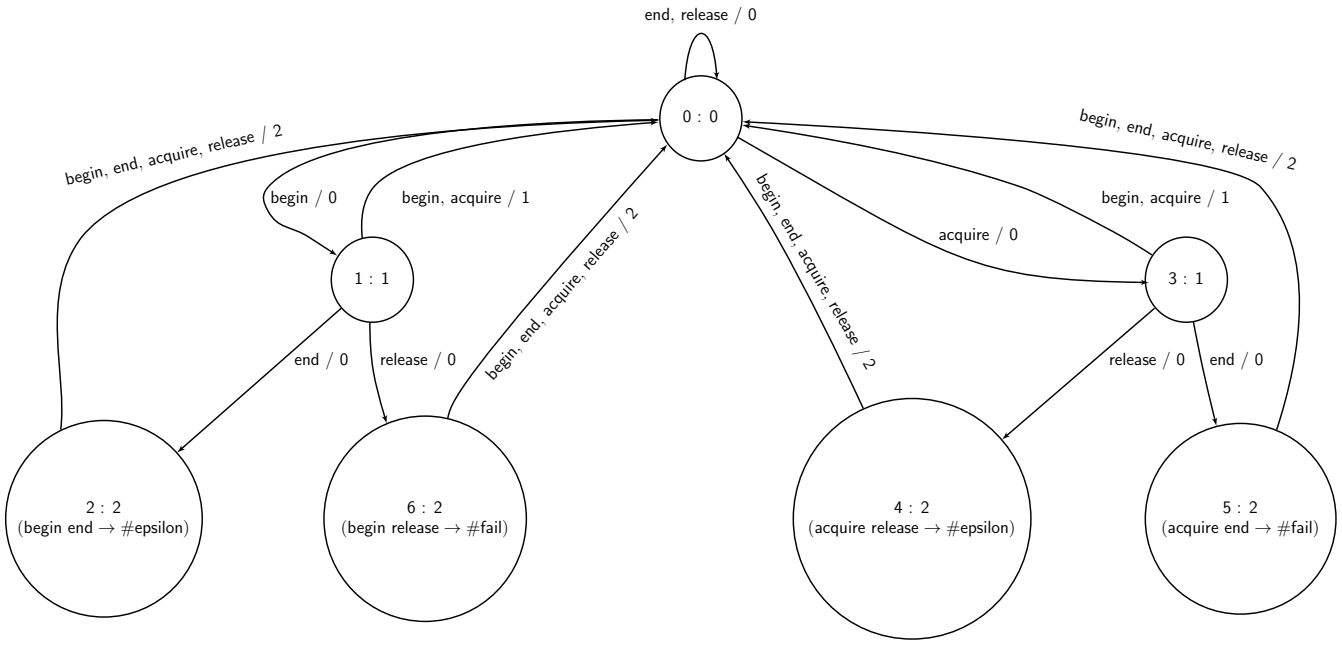

Figure 4: Pattern match automaton for the SafeLock property (see Section 1.1)

to state 0 mentioned earlier, while lines 23–27 represent a normal forward transition. 23–27 are a forward transition because the *action* of the transition is 0. As mentioned earlier, the only difference between the 0 self-transition and a forward transition is that in the self-transition the *first* SLIterator need be incremented (line 18). Lines 28–30 handle a backward transition in the PatternMatchAutomaton. As expected, with a backward transition the *first* SLIterator is incremented a number of times specified by the *action* of *transition* and *second* is *not* incremented so that the same symbol will be considered in the next iteration of the loop. One interesting property of this algorithm is that if one pattern is a prefix of another, such as the patterns "$a\ a\ \rightarrow c$" and "$a\ a\ b\ \rightarrow d$", both matches will be reported. This is undesirable behavior for rewriting because "aa" will be rewritten to c immediately and "$a\ a\ b$" should no longer be matchable. This will be accounted for in Section 3.4.

As an example of how the pattern match algorithm functions, suppose that the following series of events have been seen at a given point in a program: begin begin acquire begin end. At this point, the SafeLock property will experience its first match of a rule LHS. Figure 6 shows the state transitions as each symbol is considered, as well as the position of the *first* SLIterator. An important thing to note is that every time we transition back to state 0, the *first* SLIterator index is incremented by 1 (specified by the back transitions), and the symbol is evaluated again in state 0. In general, back transitions need not be to state 0, as we shall see. At the end of the input, the algorithm is in state 2, which matches the rule begin end → #epsilon. The *first* SLIterator correctly points to index 3, which is the last begin event. The *second* SLIterator always points at the current input, which is end. These SLIterators can then be used to quickly replace begin end with #epsilon, as we will see in Section 3.4.

3.3.2 Generating Pattern Match Automata

There are two main phases to the creation of pattern match automata. In the first phase the forward transitions of the automaton are created. In the second phase, all of the backward transitions and the self-transition that (almost) always exists in state 0 are added. During the computation of the backward transitions, the actions for the backward transition are also computed and added to the backward transitions. As mentioned, only backward transitions ever have non-0 actions, since they correspond to places in the automaton where there is a switch from matching one potential set of LHSs of rules to another. For instance, in Figure 6, between the third begin and the first acquire, there is a switch from potentially matching {begin end, begin release} to {acquire end, acquire release}, which requires no longer considering the begin event for match purposes, thus the action of 1.

To create the forward transitions for an automaton, we add one path that corresponds directly to the left hand side of each rule in our string rewriting system. We add these paths one at a time, and reuse as many states as possible. Each forward transition is assigned the action 0. Figure 7 shows the forward transitions for the pattern match automaton originally presented in Figure 4. For each LHS, we begin at state 0 and add a transition for the first symbol. Because all patterns SafeLock begin with either begin or acquire, we have only two transitions, one labeled with begin and one labeled with acquire. We continue to transitively add transitions based on the remainder of each LHS. For the two rule LHSs beginning with begin, one ends with end and the other ends with release, so there are two transitions out of state 1 labeled accordingly. As each new state is added to the machine during the forward transition phase, the depth of the state is recorded. The depth is simply the number of symbols from state 0. For instance, state 6 is at depth 2, since two symbols, begin followed by end, lead to state 6. The largest depth always corresponds to the longest rule LHS.

```
1  globals PatternMatchAutomaton pma
2  locals SLIterator first, second
3         State currentState, nextState
4         Symbol symbol
5         Transition transition
6  procedure match(SpliceList l)
7  first ← l.head()
8  second ← l.head()
9  currentState ← 0
10 while (true){
11   if (currentState.hasMatch()){
12     //signal match
13   }
14   symbol ← second.get()
15   transition ← pma.get(currentState, symbol)
16   nextState ← transition.state
17   if (nextState = 0){
18     first.next(1)
19     if (¬second.next(1)){
20       break
21     }
22   }
23   else if (transition.action = 0){
24     if (¬second.next(1)){
25       break
26     }
27   }
28   else {
29     first.next(transition.action)
30   }
31   currentState ← nextState
32 }
```

Figure 5: Pattern Match Algorithm

current state	symbol	next state	*first* index
0	begin	1	0
1	begin	0	1
0	begin	1	1
1	acquire	0	2
0	acquire	3	2
3	begin	0	3
0	begin	1	3
1	end	2	3

Figure 6: A run of the pattern match algorithm on begin begin acquire begin end

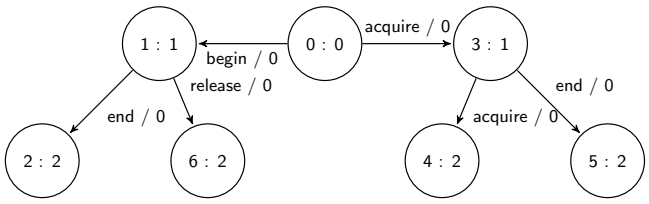

Figure 7: Forward Transitions for SafeLock (matched rules omitted)

In the second phase, the self-transition on state 0 is added first, if needed. The self-transition is only necessary if there is not a forward transition out of state 0 for every symbol used in the SRS or specified by the JavaMOP front end[6].

After potentially adding the self-transition in state 0, the backward transitions are added to the pattern match automaton. Backward transitions are only added from a given state for symbols that do not have forward transitions out of that state. All backward transitions from a given state, s, will go to the same place, so we define $fail(s) = s'$, where s' is the destination of a backward transition out of s. To find the destination for the backward transitions out of a state in pattern match automaton pma with depth d, we consider each state r of depth $d-1$ and perform the following actions, transitions are added in depth first order [1]:

1. If $pma.get(r, a)$ is a backward transition for all symbols a, do nothing.
2. Otherwise, for each symbol a such that $pma.get(r, a) = s$, do the following:
   (a) Let $s' = fail(r)$.
   (b) Compute $s' \leftarrow fail(s')$ until such point as $pma.get(s', a).action = 0$. Because state 0 must have either a forward transition or a self-transition for every symbol, such an s' must exist.
   (c) For all a' such that $pma.get(s, a')$ has no forward transition, assign $pma.get(s, a').state = s'$, ***pma.get(s,a').action = s.depth - s'.depth***.

The procedure above is essentially the same as [1]. The part in bold is specific to our algorithm for string rewriting. The action is assigned as such because the depth of a given state represents the number of symbols processed since state 0 in the automaton, thus the difference in the depths tell us the number of symbols that we need to skip with the *first* SLIterator in Figure 5. While the pattern match automaton for SafeLock has backward transitions that only go to state 0, as mentioned, this is not always the case in general. When the suffix of one LHS overlaps with the prefix of another, backward transitions that do not go back to state 0 are generated. An example of this can be seen in Figure 8, where the SRS in question is b a a → #epsilon, a a c → #epsilon. Because b a a and a a c have a suffix/prefix overlap, the backward transitions from state 3 at depth 3 go to state 5 at depth 2, resulting in an action of only 1. For example, consider input b a a c. When we switch from matching b a a to matching a a c, which occurs between states 3 and 5, we wish to only "forget" the b at the beginning, an action of 1.

3.4 Rewriting using Pattern Match Automata

The rewriting algorithm we use to monitor SRS's is presented in Figure 9. Not pictured in Figure 9, is the action of the monitor itself. As any monitoring algorithm in the MOP framework, events arrive one at a time. As each event occurs, we add it—as a symbol representing that event—to a SpliceList that contains the results of rewriting previous sequences of events. Additionally, if any rules make use of the ^ symbol, it will be added to the beginning of the SpliceList

[6]JavaMOP allows one to define events that do not appear in the specified property; these will correspond to symbols that are never rewritten by the specified SRS.

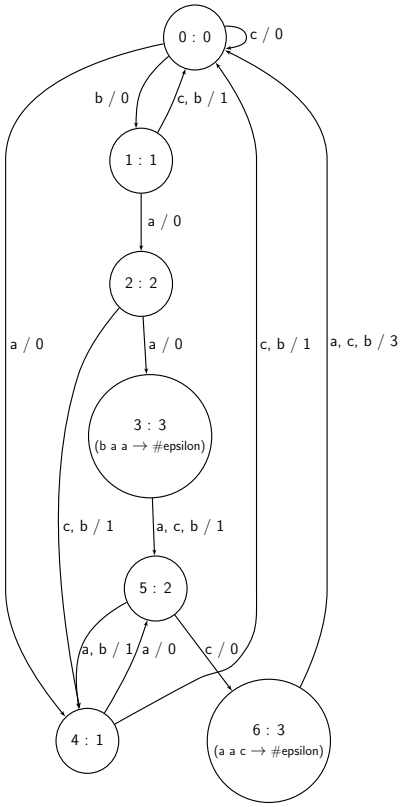

Figure 8: A pattern match automaton with overlap

and treated as a normal symbol by the rewriting algorithm. As for uses of \$, the current event must be added *before* \$[7].

After an event is added to the SpliceList, the algorithm in Figure 9 is evaluated to completion before another event can be accepted. The algorithm is similar to the pattern match procedure of Figure 5. The changes are in bold. There are three main changes: the inclusion of a loop that ensures that a normal form is reached, the actual rewriting step itself, and a section that recognizes early termination.

The first new control structure to notice is the do...while loop from line 8 to 56. This loop ensures that rewriting continues until there is a pass through the loop in which nothing changes, i.e., the string is in normal form. The new boolean variable, *changed*, controls this loop. It is set to false at the beginning of an iteration of the do...while loop, and to true on line 24, which is only executed when a rewrite occurs.

Lines 15–28 perform the actual rewriting step. The element *match* of a State contains the right hand side of the rule matched in that State. If the *match* is one of the two special keywords #succeed or #fail, a success or fail handler is executed, as appropriate, and rewriting terminates. If either handler is executed, the monitor is considered dead unless it is reset (see [27]). If *match* is something else, the splice method is called on line 22. The splice method is a special method of SLIterator that replaces a range specified by the this and an argument SLIterator with the argument sequence. Here the range is specified by *first* and *second*, and *currentState.match*

[7]Because of this there is a very small performance hit for using \$ in a rule, but ^ is essentially free.

```
1  globals PatternMatchAutomaton pma
2  locals SLIterator first, second, last
3         State currentState, nextState
4         Symbol symbol
5         Transition transition
6         boolean changed pastLast
7  procedure match(SpliceList l)
8  do {
9    first ← l.head()
10   second ← l.head()
11   currentState ← 0
12   changed ← false
13   pastLast ← false
14   while (true){
15     if (currentState.hasMatch()){
16       if (currentState.match = #succeed){
17         // raise succeed
18       }
19       if (currentState.match = #fail){
20         // raise fail
21       }
22       first.splice(second, currentState.match)
23       nextState ← 0
24       changed ← true
25       pastLast ← false
26       last ← second
27       second ← first.copy()
28     }
29     symbol ← second.get()
30     transition ← pma.get(currentState, symbol)
31     nextState ← transition.state
32     if (nextState = 0){
33       first.next(1)
34       if (¬second.next(1)){
35         break
36       }
37     }
38     else if (transition.action = 0){
39       if (¬second.next(1)){
40         break
41       }
42     }
43     else {
44       first.next(transition.action)
45     }
46     if (¬changed){
47       if (second = last){
48         pastLast ← true
49       }
50       if (pastLast and nextState = 0){
51         return
52       }
53     }
54     currentState ← nextState
55   }
56 } while (changed)
```

Figure 9: Rewriting Algorithm

event	initial l	l in normal form
begin	begin	begin
end	begin end	#epsilon
begin	begin	begin
acquire	begin acquire	begin acquire
release	begin acquire release	begin
acquire	begin acquire	begin acquire
end	begin acquire end	#fail

Figure 10: An SRS monitoring run for SafeLock

is passed as the replacement. Note that if the right hand side of the rule is #epsilon, it is represented as an empty sequence, which splice is able to handle. The splice method also correctly sets *first* and *second* to point to the beginning and end of the spliced in *match* sequence, or the next symbol if *match* was #epsilon. On line 26, we set *last* to *second*, so that *last* points to the end of the last replacement, this will be used to determine early termination. Then, on line 27, *second* is set as a copy of *first*. This ensures that segments of string which are transitively rewritten will be rewritten immediately. Because splice changes the SpliceList, it is important to set *currentState* back to state 0 because any matching will occur in the newly rewritten segment of the SpliceList.

In the last new addition to the match algorithm, from lines 46 to 53, we test for early termination of the algorithm. The idea here is to exit early if we enter a segment of the SpliceList that we know for certain cannot be rewritten. This happens when we reach a point that is past the end of the *last* SLIterator, which was set in a previous iteration, no rewrites have occurred in the current iteration, and *currentState* returns to 0. The first two requirements are fairly straight-forward: if a change occurs, new matches are possible, and if we are in a segment of the SpliceList before the last rewrite, we are still investigating symbols that are potentially new. However, if there is no rewrite in the current iteration and we are past the last change from the previous iteration, we are seeing symbols that were seen in the previous iteration with no change. The last condition, that we must return to State 0, is more subtle. The reason for this is that there could have been a rewrite in the last iteration that inserted a segment that appears in the middle of a left hand side of one of the rules. A simpler way to look at this requirement is that if *pma* is not in state 0 it is actively matching *something*. This condition for early termination can lead to an unbounded amount of saving, as the SpliceList can be of an unbounded length.

Figure 10 shows a monitor run as non-parametric events for SafeLock arrive. The non-parametric events are dispatched to the correct monitor instance by the indexing of JavaMOP (or whatever projection method is used in future language instances of MOP). The first column shows the arriving event, the second column shows the state of the SpliceList l before any rewriting, and the last column shows the normal form for l after the rewriting algorithm of Figure 9 has run. After the last event a failure has occurred, and the fail handler will execute.

4. EVALUATION

Our SRS implementation is evaluated in two contexts: first we show how it compares, within the context of JavaMOP, to finite-state logics on the DaCapo benchmark suite [7]. Then we give a comparison of our underlying SRS rewrite engine against the Maude [16] term rewriting engine, modulo associativity. The goal of the first evaluation is to show that SRS monitoring is efficient enough to be used in large programs, being not much less efficient than finite-state logics (extended regular expressions in this case). The goal of the second experiment is to show that our SRS implementation is more efficient than the state-of-the-art[8].

All experiments were performed on a machine with a 3.82GHz Intel® Core™ i7 970 hexcore with Hyper-Threading (12 hardware threads) and 24 GB of ram. Ubuntu 11.10 64 bit was used as the operating system and version 9.12 of DaCapo was used as the benchmark suite, with default inputs and the -converge option to gain convergence within 3%. OpenJDK version 1.6.0_23 as the Java virtual machine. All compiled JavaMOP specs were weaved into DaCapo using ajc 1.6.11. Maude 2.6 was used for comparison with Maude.

The following properties were used in the DaCapo experiments. The SRS versions of them (shown below) are new, while the extended regular expression versions were borrowed from [10, 9, 13, 26].

- HasNext: Do not use the next element in an Iterator without checking for the existence of it (see Section 1.1);
- SafeSyncCol: If a Collection is synchronized, then its iterator also should be accessed synchronously:

$$\begin{aligned} \text{sync asyncCreateIter} &\rightarrow \#\text{fail} \\ \text{sync syncCreateIter accessIter} &\rightarrow \#\text{fail} \end{aligned}$$

- SafeSyncMap: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner:

$$\begin{aligned} \text{sync createSet asyncCreateIter} &\rightarrow \#\text{fail} \\ \text{sync createSet syncCreateIter accessIter} &\rightarrow \#\text{fail} \end{aligned}$$

- UnsafeIter: Do not update a Collection when using the Iterator interface to iterate its elements:

$$\begin{aligned} \text{update use} &\rightarrow \#\text{fail} \\ \text{use use} &\rightarrow \text{use} \\ \text{update update} &\rightarrow \text{update} \\ \text{createIterator} &\rightarrow \#\text{epsilon} \end{aligned}$$

- UnsafeMapIter: Do not update a Map when using the Iterator interface to iterate its values or its keys:

$$\begin{aligned} \text{update use} &\rightarrow \#\text{fail} \\ \text{use use} &\rightarrow \text{use} \\ \text{update update} &\rightarrow \text{update} \\ \text{createIterator} &\rightarrow \#\text{epsilon} \\ \text{createCollection} &\rightarrow \#\text{epsilon} \end{aligned}$$

For the comparison with Maude, strings of equal numbers of 2's, 1's, and 0's, with the 2's preceding the 1's preceding the 0's were generated, and the following rewrite system applied. Note, that the language of strings that reduce to #epsilon with this rewrite system is non-context free. It is very similar to EqualityCheck from Section 1.1.

$$\begin{aligned} 1\,0 &\rightarrow 0\,1 & \qquad 2\,0 &\rightarrow 0\,2 \\ 2\,1 &\rightarrow 1\,2 & \qquad 0\,1 &\rightarrow 3 \\ 1\,3 &\rightarrow 3\,1 & \qquad 3\,0 &\rightarrow 0\,3 \\ 3\,2 &\rightarrow \#\text{epsilon} & \qquad 2\,3 &\rightarrow \#\text{epsilon} \end{aligned}$$

[8]Note that Maude is more general than our SRS engine, but there is a price for that generality, and general term rewriting makes little sense in the context of MOP event traces.

Benchmark	Original (ms)	HasNext		SafeSyncCol		SafeSyncMap		UnsafeIter		UnsafeMapIter	
		ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS
avrora	2317*	194	227	35*	103*	28	120*	253*	288*	41	134*
batik	773	0	6	5	11	9	-1	5	3	1	2
eclipse	11749	-1	-2	-2	-4	-1	-3	-2	-2	-2	-2
fop	251	922	2091	26	24	21	20	34	57	28	42
h2	3860	9	15	6	2	0	4	15	22	8	24
jython	1400	3	4	3	3	4	2	16	18	3	3
luindex	478	2	-1	2	0	0	4	1	2	0	-5
lusearch	581	1	3	-1	1	3	3	46	46	2	0
pmd	1441	27	117	139	137	10	17	72	148	177	199
sunflow	1222	5	8	0	-1	6	-3	-4	4	0	3
tomcat	1068	2	4	3	3	3	1	2	2	2	2
tradebeans	4618	2	1	-1	-3	-1	2	4	-2	-2	-1
tradesoap	3213	1	-1	1	-1	0	-2	1	0	0	0
xalan	359	5	1	5	1	6	3	90	172	7	8

Figure 11: Comparison of JavaMOP with extended regular expressions (ERE) and with the same properties expressed as string rewriting systems (SRS): average *percent* overhead (convergence within 3% except those marked with *)

N	Maude Time (ms)	SRS Time (ms)
100	42	33
1000	37038	236
5000	DNF	7112
10000	DNF	26132

Figure 12: Comparison of maude versus SRS rewrite. DNF: did not finish in one hour

Figure 11 shows a comparison of finite-state properties specified in JavaMOP using ERE and SRS. The first column shows the individual DaCapo [7] benchmarks, and the second column shows runtime of the original uninstrumented benchmarks in milliseconds. All other columns are *percent* overhead. Each benchmark-property pair converged to within 3% except the instances of avrora marked with *. The results presented for avrora that did not converge are the average of twenty runs with outliers removed, but they are still not as trustworthy as the converging results. This lack of convergence is a problem on highly multithreaded machines. We can see that even the uninstrumented, original run, fails to converge. Negative overheads are the result of noise in the experimental settings and changes in code layout due to instrumentation resulting in slightly more efficient programs.

Overall, the average overhead on the DaCapo benchmark suite was 58% for SRS, while it was 33% for ERE. When fop-HasNext—which has, by far, the worst overhead of any trial—is removed from both, the overhead drops to 29% and 20%, respectively. It must be noted, that the properties we use are specifically selected for generating large overheads; they are very intensive properties that generate *many events* (see [21]). The overhead numbers are slightly larger than reported in previous papers because we have moved to a multithreaded, and quite simply faster, machine. The monitors in JavaMOP must be synchronized, which results in higher overhead for programs that actually make use of multiple threads. Any monitoring system must do the same thing if the monitors are for cross-thread properties (like all of those properties used here). In most of the benchmark/property pairs, the performance of ERE and SRS are very comparable. For pmd-HasNext and avrora-SafeSyncMap, SRS shows more than three times the overhead of ERE, but for all other trials SRS is never more than three times worse.

Figure 12 shows the comparison of Maude to our SRS engine with the rewrite system discussed above. N refers to the number of each digit, i.e., N=100 has 300 characters in it: 100 each of 2, 1, and 0. As we can see from the results, our SRS engine runs in 78% of the time of maude at N=100. At N=1000, our SRS engine runs in .006% of the time of Maude. With larger inputs, Maude fails to complete in an hour, while our SRS engine takes less than 30 seconds on every tested input.

5. CONCLUSION

We provided the first means to efficiently monitor parametric Turing-complete specifications using string rewriting systems. By using a modified version of the Aho-Corasick string matching algorithm and a means to terminate the rewriting process early, the resultant string rewriting algorithm is quite practical, as shown in our extensive evaluation[9]. The average overhead on the DaCapo benchmark suite was 58% for SRS, while it was 33% for ERE. When the largest benchmark/property pair is removed from both, the overhead drops to 29% and 20%, respectively. A less extensive comparison of our core string rewriting algorithm with the term rewrite engine Maude, which provides implicit support for string rewriting through its rewriting modulo associativity, suggests that our approach can lead to new string rewriting engines that outperform the state-of-the-art.

[9]Special thanks to Dongyun Jin for help with DaCapo experimental settings.

6. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, pages 345–364. ACM, 2005.

[3] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA'07*, pages 589–608. ACM, 2007.

[4] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[5] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Logic Computation*, pages exn076+, November 2008.

[6] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *RV*, pages 168–182, 2010.

[7] 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'06*, pages 169–190. ACM, 2006.

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

[9] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.

[10] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549, 2007.

[11] R. V. Book and F. Otto. *String-rewriting systems*. Texts and monographs in computer science. Springer, 1993.

[12] S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *Model Checking Software(SPIN'07)*, volume 4595 of *LNCS*, pages 279–283, 2007.

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

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

[15] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *OOPSLA'07*, pages 569–588. ACM, 2007.

[16] 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.

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

[18] D. Drusinsky. Temporal Rover, 1997–2009. `http://www.time-rover.com`.

[19] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA'05*, pages 385–402. ACM, 2005.

[20] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification(RV'01)*, volume 55 of *ENTCS*, 2001.

[21] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353, 2001.

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

[24] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report `http://hdl.handle.net/2142/30006`, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[25] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'07*, pages 365–383. ACM, 2005.

[26] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.

[27] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. http://dx.doi.org/10.1007/s10009-011-0198-6.

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

[29] R. E. Strom and S. Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.