Enhancing Performance And Reliability of Rule Management Platforms

Mark Grechanik
University of Illinois at Chicago
Chicago, IL 60607
drmark@uic.edu

B. M. Mainul Hossain
University of Illinois at Chicago
Chicago, IL 60607
bhossa2@uic.edu

ABSTRACT

RulE Management Platforms (REMPs) enable software engineers to represent programming logic as conditional sentences that relate statements of facts. A key benefit of REMPs is that they make software adaptable by burying the complexity of rule invocation in their engines, so that programmers can concentrate on business aspects of highly modular rules. Naturally, rule-driven applications are expected to have excellent performance, since REMP engines should be able to invoke highly modular rules in parallel in response to asserting different facts. In reality, it is very difficult to parallelize rule executions, since it leads to the loss of reliability and adaptability of rule-driven applications.

We created a novel solution that is based on obtaining a rule execution model that is used at different layers of REMPs to enhance the performance of rule-driven applications while maintaining their reliability and adaptability. First, using this model, possible races are detected statically among rules, and we evaluate an implementation of our abstraction of algorithms for automatically preventing races among rules. Next, we use the sensitivity analysis to find better schedules among simultaneously executing rules to improve the overall performance of the application. We implemented our solution for JBoss Drools and we evaluated it on three applications. The results suggest that our solution is effective, since we achieved over 225% speedup on average.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Representation*; I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representations—*Rule-Based*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Algorithms, Performance, Experimentation

Keywords

concurrency; parallelism; rule-driven application; expert system

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2688035.

1. INTRODUCTION

Widely used in different software applications (e.g., anti-money-laundering, fraud detection, network monitoring, stock exchange trading, insurance claim management, and risk assessment) [65, 63, 53, 77], *RulE Management Platforms (REMPs)* allow software engineers to represent programming logic as conditional sentences that relate statements of facts (i.e., rules) using high-level declarative languages [14, 29, 38, 41]. New facts are inferred in REMPs automatically using *modus ponens*. An example of a rule is "if an insurance customer filed a claim for over $1Mil and this customer is 30 year old or younger, then the insurance premium should be increased by 20% for this customer." Once the facts "an insurance customer filed a claim for over $1Mil" and "this customer is 30 year old or younger" are asserted, this rule is fired (or invoked) by the underlying REMP engine and a new fact is produced, i.e., the insurance premium for this customer is increased by 20%. The popularity of REMPs is partially explained by the ease of designing and maintaining the highly modular logic of rules [35, 39, 42, 47, 59, 17, 10]. Different stakeholders can independently design and deploy rules for *Rule-driven APplications (RAPs)* that run on top of REMPs, making RAPs highly *adaptable* to frequent changes in business requirements in today's dynamic world.

REMPs have become very important in the age of *big data*—collections of large-sized data sets that contain useful patterns and rules. Programmers often encode into RAPs patterns and rules that are extracted from big data (e.g., rules that describe increases in insurance premiums for customers with some risky behavior), so that these rules are invoked in business processes. RAPs are pervasive, they are used by many major insurance companies, government agencies, and various non-profit organizations [65, 33, 38]; for example, they are used to check for fraud for 90% of all credit card transactions in the USA in real time [65]. The market for REMP engines alone is estimated to be growing at the annual rate of 10.5%, reaching $1 Billion worldwide [38].

1.1 Background on REMPs

A key property of REMPs is that they encapsulate the control flow that includes fact inference and rule firing logics, while enforcing a fundamental separation of concerns of the control flow and the rule business logic. REMPs enable software engineers to concentrate on reasoning about higher-level business logic that they encode in rules without worrying about low-level details of rule invocations by effectively delegating this job to REMP engines. With this separation of concerns, RAPs are highly adaptable to changing requirements, since stakeholders simply add new rules as independent modules to RAPs and the underlying REMP engines ensure that these rules are fired when these conditions are met.

In a model for rule-based programming, rules can be viewed as decision points in some business process [46]. Martin Fowler views a rule-based computational model for REMPs as an alternative to the imperative model where sequences of `if-then` statements with conditionals and loops are evaluated in a strictly defined order. Having many `if-then` conditions results in a hard-to-maintain and inefficient code that is not *adaptable* to frequent changes in business requirements [26]. Moreover, since the conditional expressions of `if-then` statements must be evaluated by the runtime system, some overhead is incurred. In contrast, using rules replaces `if-then` statements with an optimized network of rule invocations that are controlled by the REMP engine. In addition, burying the complexity of reasoning about executions of multiple nested `if` statements inside the REMP engine enables stakeholders to write easy-to-comprehend rules that are more efficiently executed by REMPs, since runtime evaluation of many conditional expressions is avoided. Rules are often simple and they rarely contain complex nested loops and conditional statements [30, pages 495-558]; we use this insight to offer lightweight analysis to detect and prevent data races. Many problems naturally fit this model, since their solutions are often expressed using `if-then` rules.

Using the REMP model, it is easy to maintain and evolve RAPs, since rules often do not depend on one another [26, 51, 46, 55]. Adding new rules and replacing old rules does not require recompilation of the entire RAP's source code. Stakeholders can maintain different rules independently and these rules can be added to RAPs without regard to one another. Since rules are easy-to-comprehend and highly modular, it is easy to make changes to complex RAPs to enable large organizations to modify complex business processes inexpensively, hence RAPs are highly adaptable [30, pages 370].

1.2 Drawbacks of the REMP Model

These benefits have the other side. RAPs contain highly diverse rules, and it makes their analysis very difficult. Our investigation of dozens of open-source multithreaded Java applications showed that applications spawn many threads that execute a small subset of the application's methods, and still it is very difficult to reason about even a small number of concurrently executing methods [31].

In contrast, consider a RAP at a major insurance company that has over 30,000 different rules, many of which are written by different stakeholders. Our analysis showed that at any point hundreds to thousands of rules could be fired concurrently. "No one has any idea if there are conflicting rules when a new one is added" is a comment left on a programming forums that discuss pros and cons for using rules in applications [69]. "Rules should not be used if they are strongly connected Java files" is the other comment left on a different programming forum [70]. Redhat portal specifies: "Rules engines are not really intended to handle workflow or process executions nor are workflow engines or process management tools designed to do rules" [60]. These drawbacks highlight what happen when dependencies are introduced among rules – suddenly, they lose adaptability and they become very difficult to maintain and evolve. Hiding the logic that deals with these dependencies inside the REMP engine is a way to free stakeholders from dealing with the accidental complexity of explicitly coded dependencies among rules, thus ensuring adaptability of RAPs.

1.3 The PAR Model

A fundamental problem of REMPs that we address in this paper lies at the intersection of *Performance, Adaptability and Reliability (PAR)* that is shown in Figure 1. Recall that in today's rapidly changing business requirements, software adaptability is a critical element that ensures success [71, 16, 45, 25]. For example, an

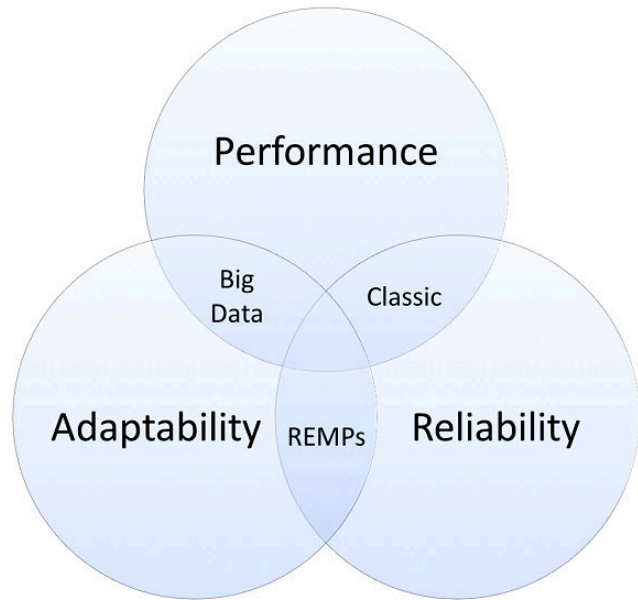

Figure 1: The PAR model.

efficient but inflexible software application makes it very difficult for businesses to refocus their effort on new revenue-generating opportunities. REMPs provide software adaptability by burying the complexity of rule invocation in REMP engines and enabling stakeholders to concentrate on business aspects of highly modular rules. Unlike RAPs, in "classic" software applications, modularity and adaptability are often sacrificed while optimizing these applications for performance and reliability [8]. With advent of big data and cloud computing, the focus shifted somewhat towards the intersection of adaptability and performance while sacrificing reliability, where applications are created from modular components (e.g., RapidMiner, Weka) that use machine learning and data mining algorithms to compute somewhat incorrect and approximate results fast and if needed, quickly change the configurations of these applications. New methodologies known as probabilistic programming and cloud accelerators put emphasis on computing approximate results faster and applying quick program repair techniques to allow buggy programs to complete calculations at the expense of reliability [23, 21, 22, 11, 52, 61, 64, 18].

Naturally, RAPs are expected to have excellent performance, since REMP engines should be able to invoke highly modular declarative rules in parallel in response to asserting different facts [75, 32, 30]. In some exceptional cases it is straightforward to do so, e.g., checking for credit card transaction fraud is done by invoking rules in parallel, since they almost never produce any side effects. However, in general, it is difficult to parallelize executions of rules.

Early languages for REMPs were purely declarative [41], however, over years, tight integration of RAPs with legacy systems made vendors mix imperative and declarative constructs [38, 53, 65, 33]. For example, programmers can define facts by assigning values to global variables. Prominent examples of open source REMPs with mixed languages include CLIPS [76], JESS [39] and JBoss Drools [5, 13], where C functions and Java methods are used in rules (see an example in Figure 2). Commercial REMPs with mixed language constructs include BizTalk by Microsoft [36] and Fusion by Oracle [27]. These and other REMPs fire rules sequentially [67], e.g., Oracle Fusion documentation states: "Rules fire sequentially, not in parallel. Note that rule actions often change the set of rule activations and thus change the next rule to fire" [58]. Thus, the results of the computations depend on the order of rule executions, i.e., if executing rules in parallel may lead to different

results for the same input values (i.e., facts) and for the same environment configuration, hence the loss of reliability.

Not only do many REMPs execute RAPs sequentially, but also programmers are often restricted from using locks to handle concurrent accesses to resources from different RAPs to prevent races. Locks introduce complex dependencies among rules, thereby defeating the separation of concerns and eventually the adaptability of RAPs [56]. For example, waiting on locks to be released overrides the control flow computed by the underlying REMP engines. Clearly, programmers should be able to write their code for rules without worrying about races, and REMP engines should take care of rule firing and preventing data races at the same time. Thus, *a fundamental problem of REMPs is how to enhance the performance of RAPs without sacrificing their adaptability and reliability*, i.e., to move REMPs into the center of the PAR area in Figure 1.

1.4 Our Contributions

Our novel solution for enhancing *PErformance and Reliability for ruLe-driven ApplicaTiOns (PERLATO)* connects separate layers or REMPs in a way that enable us to solve the fundamental problem of REMP. First, we obtain a rule execution model from a RAP that approximate different execution scenarios by using the `if-then` structure of rules by analyzing their antecedents and consequents. To do that, we statically analyze conditions in antecedents of rules and possible ways that rules can be triggered by approximating the control flow in consequents. As a result of this analysis, we obtain constraints that can be solved using constraint solvers to obtain dependencies among rules. Second, the obtained rule execution model is used in PERLATO to detect races statically among these rules effectively and efficiently. A key idea is that if the rule execution model shows that some rules cannot run concurrently (e.g., one rule is triggered based on the constraint $x > y$ and the other rule is triggered based on the constraint $x < y$, where x and y some variables), the complexity of the data race analysis can be significantly reduced.

Next, the rule execution model and locking strategies for a given RAP are passed to the REMP engine, so that it can precompute an execution schedule for rules in a RAP to optimize the performance of the RAP. This is the essence of a *cross-layer design* where we pass the information that we compute at the application layer deep into the REMP engine layers to schedule rules in a way that lets faster executing rules proceed sooner and a longer executing rule to wait until other faster executing rules finish, so that some performance can be gained by reducing an average waiting time. Using a scheduler to enforce a specific order of rule execution is our novel way to optimize the performance of the RAP that addresses the issue of reliability, since the REMP engine will enforce locking and scheduling leading to the same results of executions for the same input data and environment configuration. This paper makes the following contributions.

- With PERLATO, we show how to parallelize rule execution automatically inside REMP engines without requiring programmers to use locks. Our tool and results are available at `http://www.cs.uic.edu/~drmark/perlato.htm`.
- Using sensitivity analysis [62], we show how to compute *symbiotic* schedules (i.e., co-scheduling conflicting jobs to achieve higher speedup [24, 68]) of execution of rules that have concurrent accesses to resources.
- We implemented PERLATO for JBoss Drools, an open-source enterprise-level REMP [5, 13] and we evaluated PERLATO on three RAPs. The results suggest that PERLATO is effective and efficient, since we achieved up to 225% speedup on average without observing any races.
- Summarily, we extend the theory of REMPs by enhancing their performance and reliability while preserving adaptability of RAPs without violating the separation of concerns between REMP engines and high-level rules.

```
rule "Rule-Credit" salience 10                              1
when                                                        2
    $cashflow : Cashflow( $account:account,                 3
    $date : date, $amount : amount,                         4
    type==Cashflow.CREDIT )                                 5
    not Cashflow(account==$account,date<$date)              6
then                                                        7
    //some code                                             8
    $account.setBalance(                                    9
        $account.getBalance()+$amount);                     10
    retract($cashflow);                                     11
end                                                         12
rule "Rule-Debit" salience 1                                13
when                                                        14
    $cashflow : Cashflow( $account : account,               15
    $date : date, $amount : amount,                         16
    type==Cashflow.DEBIT )                                  17
    not Cashflow(account==$account,date<$date)              18
then                                                        19
    //some code                                             20
    if($account.getBalance()>$amount){                      21
        $account.setBalance(                                22
        $account.getBalance()-$amount); }                   23
    else { new BlockedAccount($cashflow); }                 24
    retract($cashflow);                                     25
end                                                         26
```

Figure 2: An example of debit and credit rules from a banking RAP that concurrently modify an account balance.

2. THE PROBLEM

In this section, we provide a motivating example, discuss interplays between the components of the PAR model, and give the problem statement.

2.1 Motivating Example

Consider account debit and credit rules[1] that are shown in Figure 2 – they are similar to over 30,000 rules that an insurance claim handling RAP comprises at a major insurance company. The headers of the rules that are located in line 1 and line 13 contain the names of these rules, `Rule-Credit` and `Rule-Debit` correspondingly, and their *saliences*, with which programmers define priorities that the REMP engine should give to rules when choosing to execute them. The design idea of specifying the value of salience equal to 1 for rule `Rule-Debit` versus value 10 for rule `Rule-Credit` is to ensure that the REMP engine will first deposit money to the account before it subtracts an amount when both rules are triggered, thus preventing overdrafts in some cases.

Suppose that a bank user debits her account and a web store credits this account at the same time. Correspondingly, two `Cashflow` objects will be created that will trigger these rules by matching antecedents in the `when` parts of these rules in lines 2–6 and lines 14–18. The condition `not Cashflow` ensures that there is no

[1] `http://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch09.html`

`Cashflow` object with an earlier date. In the end of the consequence part of the rule, the object `Cashflow` is retracted in line 11 and line 25 to ensure that it will not trigger rules any more. To handle overdrafts, the object of type `BlockedAccount` is created in line 24, and this fact will trigger some rules that handle overdrafts. Declarations of all classes are not shown here for simplicity.

Since JBoss Drools engine executes rules sequentially, there are no concurrency bugs in this example. However, doing so worsens the performance of RAPs. Code in lines 8, 11, 20, 24, and 25 can be executed in parallel, since locks are required only for the method `setBalance` in lines 9–10 and 21-23 to prevent a data race to update the value of the concurrently modified variable `account`. While it seems straightforward to parallelize the execution of rules and let programmers use locks, it would lead to serious problems.

2.2 Parallelism Interferes With Saliences

Since REMP engines execute rules sequentially, there are no concurrency bugs in our motivating example, but the performance of the RAP is much worse when compared with unrestricted parallel execution of rules. In a gedanken experiment, let us assume that a REMP engine executes rules in parallel, (e.g., one rule per thread) and lock objects are used to synchronize concurrent accesses. Since code in different rules (i.e., threads) locked by the same object cannot interleave, one thread will execute and the other thread will suspend execution until the first thread release the lock. However, this standard practice in multithreaded programming leads to serious challenges when applying it to RAPs – dependencies are introduced among different rules, leading to lost adaptability and making it very difficult to maintain and evolve RAPs.

Parallelizing rules and using synchronization locks interfere with salience values. Consider the situation when conditions of two or more rules are satisfied. These rules are fired, and during executing these rules more facts are asserted and `when` conditions of rules are satisfied, which leads to firing other rules until there are no more rules whose conditions are satisfied. The set of rules whose conditions are satisfied at any given time is called the *conflict set*. REMP engines employ different strategies for conflict resolution [41, pages 85-87], most popular of which are *random*, where rules are chosen to fire at random and *recent*, where rules are ranked higher if they use data that have been most recently created or modified in memory. These strategies work in conjunction with the values of salience, which programmers specify for some rules, and there lies a challenge.

Consider our example in Figure 2 and suppose that both rules are triggered at the same time. According to the higher value of salience, the rule `Rule-Credit` should be give a higher priority (e.g., by giving it a larger time slice to execute) by JBoss Drools (i.e., the REMP engine) before the rule `Rule-Debit`. However, the lock object may be reached faster in the rule `Rule-Debit`, after which the rule `Rule-Credit` will be put on hold to wait until the object is released. In this scenario, using a lock object effectively overrides the intention of the programmer to give the priority to the rule `Rule-Credit`. Moreover, a new `BlockedAccount` object that is created in line `17` may trigger more rules in the RAP that otherwise will not be triggered if the order of executing these rules would be different, i.e., no account overdraft occurs. Given the large number of possible interleavings among tens of thousands of rules in a RAP, it is very difficult to reason about interactions between saliences and synchronization lock mechanisms. Therefore, it is not enough to introduce locks to prevent races, rules must be scheduled in a way to preserve priority invariants that are embedded into saliences of these rules.

2.3 Reliability Meets Performance In REMPs

Loss or reliability of RAPs comes from two sources: different orders in which rules are executed by the REMP engines for consecutive runs of the same RAP with the same input facts and races between parallelized executions of rules. Recall from Section 1.3 that the Oracle Fusion documentation warns that rule actions often change the set of rule activations and thus change the next rule to fire. It is contrary to the classic example in the parallelism and concurrency theory, where two or more threads concurrently execute the same set of instructions (e.g., increment a variable) and as long as locks are applied correctly w.r.t. the atomicity assumptions, the result of the execution is always the same for the same input data. Adjusting this example to REMPs, we view a rule as a thread and these rules/threads execute different sets of instructions, some of which may spawn additional threads of execution based on the order in which program variables are assigned values in different threads. Reasoning about such complicated scenarios is difficult.

We studied two large insurance systems where REMP engines were allowed to execute rules in parallel. Performance is paramount for these systems and the reliability may be of lesser importance (e.g., a precise oracle is not known for an insurance quote or risk assessment due to the time-dependent nature of input data). Sometimes, it is more important that a RAP computes an approximately correct result fast like in the case of determining if a credit card transaction is fraudulent [12]. Since the user does not know what the correct result should be (e.g., a precise insurance quote), the user does not perceive any loss of reliability for a slightly different result. *However, when the system produces different results consecutively for the same computational task using the same input data, it is a serious problem, since it reduces the confidence of the users in the RAP and it impacts negatively their perception about the reliability of the RAP.*

Consider a situation where the user obtains an insurance quote from a RAP of some insurance company, then the user shops around, compares different quotes and comes back to the RAP from that insurance company to purchase the insurance. This time, when entering the same data the user will get a different quote. The loss of reliability comes from the fact that the execution order for different instructions can be affected by multiple factors beyond the control of stakeholders: input/output loads that lead to high variability in times that some instructions take to execute, the CPU load, RAM fragmentation and the frequency with which garbage collector is run, essentially, running other concurrent processes that steal CPU cycles and make RAP instructions execute longer. All in all, sometimes even slight changes in the non-functional parameters of the environment (e.g., paging on demand) for executing RAPs result in different orders of instruction interleavings that lead to different results, hence the loss of reliability. It is our goal to ensure reliability while parallelizing the execution of rules.

Even if locks could be used, it is difficult for programmers to make correct atomicity assumption and deal with *unserializable* executions, i.e., a property for the concurrent execution of several operations where their effect is not equivalent to that of a serial execution of these operations [49, 43, 48]. In our motivating example in Figure 2, it may be relatively easy to see that the code between lines `21–23` should be treated as atomic; however, for tens of thousands of rules that are written by different programmers, it is difficult to determine correct atomicity assumptions. Recall from Section 1.2 that stakeholders often do not have an idea how rules interact in a RAP. It is highly likely that even if programmers used synchronization locks in RAPs, which would defeat their adaptability, races would still remain and RAPs will not be reliable.

2.4 The Problem Statement

Our goal is to achieve with a high degree of automation the following multiple conflicting objectives: 1) enable REMPs to execute rules in RAPs in parallel; 2) do not violate the separation of concerns in REMPs by requiring programmers to use synchronization lock mechanisms for concurrent accesses to shared resources; 3) prevent races in parallelized RAPs without explicit using of locking mechanisms by programmers; 4) enable reliable executions by guaranteeing that the same RAP outputs the same values for the same input facts under the same hardware and software configurations, and 5) choose an effective schedule for executing rules that concurrently access the same resources to improve the overall performance of RAPs.

We aim to make PERLATO sound, i.e., to ensure the absence of races. It means that our solution should be conservative, since it assumes that all accesses to external resources (e.g., network, files, databases) are concurrent, and all statically unresolved references to variables in some rule are considered concurrent with all unresolved references to resources in other rules. At the same time, the execution of RAP should be parallelized to achieve a speedup when compared with the baseline approach when executions of all rules are sequentialized. In general, it is an NP-complete problem to detect all races [15]. There are many approaches for enabling race detection and protecting shared resources automatically using locks [9]. Once read-write and write-write concurrent accesses to resources among rules are identified, these approaches can be used to define synchronization of rules around these concurrent accesses.

It is not our goal to use these approaches in PERLATO, but to abstract them to study how much they can contribute to improving performance and reliability of RAPS. Since we use a sound conservative approach that leads to false positive conflicts, too many of them may lead to less than optimal parallelization strategies for RAPs. Our insight is that a point of attack is based on using a rule execution model that can eliminate many infeasible scenarios when rules cannot be instantiated concurrently based on contradictions among their antecedents.

3. OUR SOLUTION

In this section, we discuss key ideas of our solution, explain the architecture of PERLATO, give an algorithm for synchronizing concurrent accesses to resources in REMPs, and describe how we select schedules for parallelization of rules.

3.1 Key Ideas

Our key idea is threefold. First, we obtain a rule execution model, which is an overapproximation of the actual behavior of RAP. Our goal is to determine rules with concurrent accesses to the same resources that cannot be executed at the same time due to contradictions in their antecedents. Using this model, read-write and write-write concurrent accesses to resources are obtained from rules in RAPs statically. Naturally, there will be false positives, since our analysis can miss contradictions in rule antecedents. Since our goal is to make PERLATO a sound approach, we will make conservative approximations about read-write and write-write concurrent accesses to resources among rules, which may negatively affect the speedup that is achieved by parallelizing the execution of rules. However, as we show in Section 4, we achieve significant speedup even with this conservative approach.

We consider three abstract levels to synchronize rules based on the scope: rule-level, atomic section level, and a single operation which accesses a resource (or variable) concurrently. Rule-level synchronization has the coarsest granularity – the entire rule is locked by a REMP when executed and other conflicting rules wait

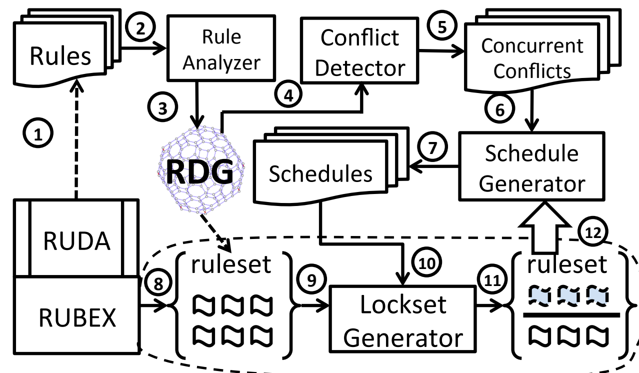

Figure 3: PERLATO's architecture and workflow.

upon completion of the executing rule. With atomic synchronization, a REMP places the lock before the first conflicting operation for a rule and holds this lock until the last conflicting operation is executed. Finally, the finest granularity of synchronization is when the lock is acquired before a conflicting operation access a shared variable and released right after it is executed. We experiment with these synchronization levels and we show in Section 4 that speedup is significant between rule and atomic synchronization levels.

The third part of our idea is to schedule executions of rules around synchronization locks, so that the same results will be consistently outputted if the same RDA is executed with the same input values under the same hardware/software configuration. That is, we impose a complete order among all rules that are fired and are in the working memory (i.e., a special memory region in REMPs where fired rules are placed for execution) that have conflicts with one another and this order is imported and used by REMPs for subsequent executions of the RAPs. We show in Section 4 that ordering rules with read-write and write-write concurrent accesses to resources guarantees the consistency of output results and achieves a speedup of up to 25% for atomic level synchronization level when the same RAP is executed with the same input values under the same hardware/software configuration.

3.2 PERLATO Architecture

The architecture of PERLATO is shown in Figure 3. Solid arrows show command and data flows between components, dashed arrows show relations between components and the block arrow with the label (12) indicates a feedback loop; numbers in circles indicate the sequences of operations in the workflow. The input to PERLATO is the set of rules of the RAP, and it is specified with the dashed arrow labeled (1). The RAP is hosted and run on top of a REMP. The output of PERLATO is the scheduled rule set that is specified with the arrow labeled (11).

We use a lightweight static analysis for detecting read-write and write-write concurrent accesses to resources among rules. Recall from Section 1.1 that rules are often simple and they rarely contain complex nested loops and conditional statements; in addition, recall from Section 1 that only few rules access resources concurrently. The latter makes sense; too many dependencies among rules defeat a goal of using a REMP in the first place, since stakeholders will not be able to obtain benefits that we discussed in Section 1.1. Therefore, our insight is to use lightweight context-insensitive interprocedural analysis to resolve references to objects and their fields. To do that, we construct and traverse a *control-flow graph (CFG)* of the RAP. When traversing the CFG we obtain a list of all objects and their fields that are referenced in rules. We perform virtual-call resolution using static class hierarchy analysis, and we take a conservative approach by accounting for all references of

methods that can potentially be invoked. We also automatically assign concurrent accesses to all unresolved references including network calls, file and database accesses. This conservative approach catches all races, but it also produces false positives that will reduce the performance of PERLATO. However, even with this conservative baseline the performance of PERLATO improves the state-of-the-art significantly as we show in Section 5.

As part of the lightweight static analysis, the Rule Analyzer (2) inputs rules and then produces (3) a rule execution model which (4) is used by the Conflict Detector (5) to obtain read-write and write-write concurrent accesses to resources, i.e., Concurrent Conflicts, which are the input (6) to the Schedule Generator that uses a race detection algorithm (7) to output Schedules for synchronization locks for handling these concurrent accesses within the REMP engine. This information is buried within the REMP engine thus preventing additional accidental complexity.

The dynamic phase of PERLATO starts with executing the RAP on top of the REMP where some facts are asserted and (8) a rule set is computed that includes the rules that are triggered by the asserted facts. At this point, this rule set is supplied (9) into the Lockset Generator along with (10) the precomputed Schedules that uses a locking strategy to output (11) a partitioned rule set (it is shown with the thick horizontal line that separated rules with dashed line border on top and solid border rules on the bottom). This top section of the rule set contains rules that can be executed in parallel and the bottom section contains rules whose execution order is defined based on the precomputed schedule. At this point, REMP continues to execute rules of the RAP. The REMP engine periodically applies sensitivity analysis [62] during RAP maintenance phases (e.g., regression testing) to perturb schedules and determine a better one that can improve the performance thus (12) realizing a feedback loop.

3.3 PERLATO Lockset Algorithm

Our algorithm for generating execution schedules and synchronization locks is shown in Algorithm 1. Our goal is to abstract existing race detection algorithms, so that we can roughly evaluate the bounds in performance improvements that these algorithms can give. Algorithm 1 consists of two procedures: the procedure `ComputeLockset` that determines sets of locks for each pair of rules with read-write and write-write concurrent accesses to resources and the procedure `AtomicLocks` that computes atomic syncronization locks for all pairs of rules with concurrent accesses. We give the algorithm only for atomic levels because rule-based and shared variable-based lock levels are trivial to compute.

The algorithm's procedure `ComputeLockset` takes as its input the set of all rules in the RAP with read-write and write-write concurrent accesses to resources, $\mathcal{R}$ and it outputs sets of locks for each pair of rules, (L^s_{ij}, L^e_{ij}). A key idea of this procedure is to compute the set of lock starts, L^s_{ij} and lock exits, L^e_{ij} for the rule r_i assuming that it executes concurrently with some other rule, r_j. This algorithmic procedure consists of two nested loops between Lines 3–14. For each pair of rules, r_i and r_j the set of conflicts is computed in Line 6 where it is placed in the variable V_{ij}. If at least one of these rules accesses a shared variable with the write access (see Line 7), we compute the locations of accesses to these variables within the rule in Lines 8–12. Correspondingly, we update the values of (L^s_{ij}, L^e_{ij}) for these rules. Once the loops finish, the procedure terminates. As a results, all locks are computed conservatively for all combinations of read-write and write-write concurrent accesses in rule pairs.

The algorithm's procedure `AtomicLocks` in Line 17 that takes as its input the set of rules for which lock sets are computed by the procedure `ComputeLockset`, $\mathcal{R}$ and it outputs sets of atomic locks for each rule, r_i, L^s_i and L^e_i and the sequence, $\mathcal{S}$ for entering atomic sections protected by lock sets for rules with concurrent accesses to resources. Lines 18–32 in Algorithm 1 specify the body of the procedure `AtomicLocks`. For each rule, the lock indicators L^s_i and L^e_i are initialized in Line 19. While iterating through all rules in the triggered rule set, $\mathcal{R}$ in Lines 20–31, the procedure computes the minimum size atomic section of shared variables that that are concurrently accessed by other rules and this section will be protected by synchronization locks within the REMP engine. We sort rules in a random order; precedence is given to rules with higher values of salience; among rules with the same salience we sort them in a random order and assign unique sequence integers to them and we use these integers to add rules to the sequence set $\mathcal{S}$ in Lines 27–29 that is used in a rule scheduler within the REMP engine. The algorithm terminates when all rules are explored.

Algorithm 1 Obtaining atomic synchronization locks for rules.

```
1: ComputeLockset( Set of rules R )
2: V ← ∅ {Initialize the set of conflicts}
3: for all r_i ∈ R do
4:    L^s_ij ← ∅, L^e_ij ← ∅ {Initialize the values of lock sets for each
      pair of rules to empty. L^s designates where the lock is placed
      and L^e where it ends in the rule r_i when it is executed with
      the rule r_j.}
5:    for all r_j ∈ R ∧ r_i ≠ r_j do
6:       V_ij ← conflict(r_i, r_j)
7:       if access(V_ij, r_i) = W ∨ access(V_ij, r_J) = W then
8:          if loc(r_i, V_ij) < loc(r_i, L^s_ij) then
9:             L^s_ij ← V_ij
10:         else if loc(r_i, V_ij) > loc(r_i, L^e_ij) then
11:            L^e_ij ← V_ij
12:         end if
13:      end if
14:   end for
15: end for
16: return (L^s_ij, L^e_ij)
17: AtomicLocks( Set of rules R )
18: for all r_i ∈ R do
19:    L^s_i ← END_OF_RULE_i, L^e_i ← START_OF_RULE_i
20:    for all r_j ∈ R ∧ r_i ≠ r_j ∧ conflict(r_i, r_j) ≠ ∅ do
21:       if L^s_i > loc(r_i, L^s_ij) then
22:          L^s_i ← L^s_ij
23:       end if
24:       if L^e_i < loc(r_i, L^e_ij) then
25:          L^e_i ← L^e_ij
26:       end if
27:       if r_i ∉ S then
28:          S ↦ S ∪ r_i
29:       end if
30:    end for
31: end for
32: return S, L^s_i, L^e_i
```

3.4 Achieving Reliability By Determining And Enforcing Schedules In REMPs

Recall the example from Section 2.3 that shows the loss of reliability in REMPs, where the user of an insurance RAP runs it twice with the same inputs and obtains a different quote. Our idea is to determine execution schedules for conflicting rules for RAPs, so that rule saliences are respected.

Our idea of computing optimal execution schedules to enforce the same results for executing the RAP with the same input data is based on our observation that a REMP engine usually hosts one RAP, since industrial RAPs are big and mission critical, and it is impractical to run more than one such RAP on top of a REMP engine. In contrast, the schedulers of operating system kernels do not preempt processes to avoid race conditions, since it is not feasible to find an optimal context switching schedule quickly for multiple processes and the overhead of doing it is prohibitive. However, in the case of REMPs it is possible to precompute an execution schedule for a RAP using our rule execution model and the REMP engine will use it to both improve the performance and guarantee reliability without requiring stakeholders to change their existing programming practices.

THEOREM 1. *If the RAP is executed two or more times with the same input values and the same environment configuration, then PERLATO preserves the output values for this RAP.*

PROOF. The proof is by contradiction. Let us assume that two executions of the same RAP using the same starting states produced different output values. Since the starting state is the same, it would mean that the execution orders $\mathcal{S}^m \neq \mathcal{S}^{m+1}$ or the interleavings of instructions are different for executing rules. However, race detection algorithmic procedures from Algorithm 1 guarantee that every possible conflict is protected by locks, thus preventing instruction interleavings within atomic sections that are protected by the same locks. Moreover, the scheduler will ensure that the order of rule execution is always the same for the same starting states for the same RAP. Therefore, the same values are produced for both executions leading to the same output values, hence the contradiction. □

3.5 Symbiotic Scheduling

Recall from Section 1 that *symbiotic* scheduling is co-scheduling conflicting jobs to achieve higher speedup of execution of rules that have concurrent accesses to shared resources. Symbiotic scheduling is widely used in the schedulers of simultaneous multithreading processors [24, 68]), and we apply it in PERLATO to determine if it is possible to improve the performance of RAPs over time.

Our idea is the following. Using sensitivity analysis [62], we permute the order that is computed in Algorithm 1 in which conflicting rules are scheduled to access resources concurrently using synchronization locks. For our motivating example that is shown in Figure 2 we will change the order of rules `Rule-Debit` and `Rule-Credit`. A goal of this operation is to obtain samples of alternative orders of executions that may give us clues if scheduling of rules can be changed to improve the overall performance of the RAP. Of course, this exercise should be done independently from live production execution of RAPs, so that the consistency of the output values will not be violated. We suggest that this exercise can be done as part of acceptance testing of RAPs. We experiment with symbiotic scheduling and show that the overall performance of RAPs can be improved by up to 20% in Section 5.

4. EXPERIMENTAL EVALUATION

In this section, we pose research questions (RQs), explain our methodology and variables and discuss threats to validity.

4.1 Research Questions

We seek to answer the following research questions.

RQ1: Is PERLATO effective in achieving higher speedups?

RQ2: Is finer granularity locking strategy more effective in obtaining higher speedup for RAPs?

RQ3: Is symbiotic scheduling effective in obtaining higher speedups?

The rationale for RQ1 is to compare the elapsed time it takes to execute RAPs using the baseline approach with the original JBoss Drools sequential rule execution engine to the elapsed time it takes to execute these RAPs using parallel versions of JBoss Drools. Our goal is to show that PERLATO is more effective than this baseline approach. The rationale for RQ2 is determine if finer granularity locking leads to much higher speedups. Recall that PERLATO allows three types of locking strategies: **R**ule-level locking were all concurrent accesses in a given rule are protected using synchronization lock objects as the rule is about to execute and released only after the REMP engine finishes executing the rule; **A**tomic locking that is shown in Algorithm 1 where the section of the rule code is locked starting with the first concurrent access and this lock is released with the last concurrent access; and finally, the resource- or **V**ariable-level locking where each operation that uses concurrently shared variables is surrounded with a lock/unlock commands. Using rule-level locking is the simplest locking strategy, but it may lead to unnecessary waits by other rules with concurrent accesses; and the variable-level locking strategy seems to be most efficient, but it may leads to the increased frequency of deadlocks. Answering RQ2 will help programmers select an optimal locking strategy. Finally, the rationale behind RQ3 is to determine if symbiotic scheduling results in higher speedups for RAPs versus the baseline approach where the schedule is computed by Algorithm 1.

4.2 Subject Applications

We evaluate PERLATO on three RAPs, each of which has been developed by five to eight graduate students as part of their master project work and taking a graduate course on distributed object programming at the University of Illinois at Chicago. Characteristics of the subject RAPs are shown in Table 4. The number of concurrently accesses variable is small, and it is expected to be small, since it is an indication of good rule design. Each subject RAP uses a database, and we report information about databases in the last three columns. With our conservative static analysis the numbers of false positive are large percentagewise, even though their absolute values are small. It is more important to determine how they affect the speedup that can be achieved with parallelizing the JBoss Drool engine.

4.3 Methodology

We aligned our methodology with the guidelines for statistical tests to assess randomized approaches in software engineering [2]. Since parallel execution with different input facts may result in different speedups, several executions with different inputs for subject RAPs are required to answer the RQs. Our goal is to collect highly representative samples of runs when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Since our experiments involve the probability of obtaining different speedups when executing subject RAPs with different input facts, it is important to conduct the experiments multiple times to pick the average to avoid skewed results. For each subject RAP, we ran each experiment 50 times with each approach to obtain a good representative sample.

Our first set of experiment is to obtain baseline indicators of the performance of the sequential and fully parallelized JBoss Drools engine. Recall that the state-of-the-art implementation is sequential and a fully parallelized version does not use any synchronization locks. Even though the latter implementation does not prevent any races, it gives us an upper bound on the performance of RAPs.

The second set of experiments involve using synchronization locks at the **R**ule, **A**tomic (computed using Algorithm 1), and **V**ariable

Table 1: Characteristics of the subject applications: their code name and the full name are shown in the first and the second columns respectively followed by the lines of code, *LOC* column. The next column, *Rules*, shows the total number of rules followed by the total number of shared variables, S_{var} that may be accessed concurrently from different rules. The next column shows the total number of false positives, *FPs* detected through static analysis, then the maximum number of rules, R_{max} that concurrently accessed at least one variable and the minimum number of rules, R_{min} that concurrently accessed at least one variable. Last three columns show the number of tables in the database that the RAPs access, the number of rows and the number of columns in those tables.

Application	*Name*	*LOC*	*Rules*	S_{var}	*FPs*	R_{max}	R_{min}	*Tables*	*Rows*	*Cols*
EEWS	Early Epidemic Warning System	4,029	13	8	4	6	2	6	1,005,918	27
TAXC	TAX Calculator	13,215	23	5	2	8	2	28	179,372,032	89
IMS	Insurance Management System	17,249	79	5	1	33	27	10	1,076,550	78

levels. Intuitively, we expect that the performance of RAPs should be better than the one of the sequential rule execution but lower when compared to the fully parallelized JBoss Drools engine.

For symbiotic scheduling, we perform 50 runs for each set of input facts using different permutations of schedules for each run, and then we compute the average elapsed execution time and the variance for each run from the average execution time. Then, we select a schedule that has the least sum of variances for different sets of input facts.

4.4 Variables

We have three independent variables: the subject RAP, the approach (DROOLS or PERLATO), and sets of input facts for each RAP. For the approach DROOLS we have two types of experiments: sequential and parallel version of the engine. For PERLATO, there are three types of experiment: the **R**egular or baseline, where a subject RAP is run using the rule-level synchronization, the experiment with the **A**tomic-level synchronization that enables concurrent execution of unsynchronized parts of rules from subject RAPs, and finally, the experiment with **V**ariable synchronization that enables RAPs to execute with the highest concurrency.

We measure the performance of RAPs as the elapsed execution time, and it is a dependent variable. Using its values we obtain speedups for PERLATO when compared with the baseline sequential JBoss Drools engine. We repeated each experiment 10 times. Thus, the total number of experiments is equal to three RAPs $\times$ (three types (R,A,V) for PERLATO + two types (Seq and Par) for DROOLS) $\times$ three sets of input facts $\times$ 50 times = 2,250 experiments. We report statistical results.

4.5 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for speedups for different approaches. We seek to evaluate the following hypotheses at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in the values of speedup between R, A, and V approaches for all subject RDAs.

H_1 An alternative hypothesis to H_0 is that there is statistically significant difference in the values of speedups between R, A, and V approaches for all subject RDAs.

Once we test the null hypothesis H_0, we are interested in the directionality of means, μ, of the results of control and treatment groups. We are interested to compare the effectiveness of the R, A, and V approaches vs one another.

H1 (Speedup of R versus A). The effective null hypothesis is that $\mu_R = \mu_A$, while the true null hypothesis is that $\mu_R \leq \mu_A$. Conversely, the alternative hypothesis is $\mu_R > \mu_A$.

H2 (Speedup of R versus V). The effective null hypothesis is that $\mu_R = \mu_V$, while the true null hypothesis is that $\mu_R \leq \mu_V$. Conversely, the alternative hypothesis is $\mu_R > \mu_V$.

H3 (Speedup of A versus V). The effective null hypothesis is that $\mu_A = \mu_V$, while the true null hypothesis is that $\mu_A \leq \mu_V$. Conversely, the alternative is $\mu_A > \mu_V$.

The rationale behind the alternative hypotheses to `H1–H3` is that finer granularity locking may lead to higher speedups. These alternative hypotheses are motivated by our belief that finer granularity locking may not necessarily lead to statistically significant increase of speedups, given the complexity of even small RAPs and dependencies among their rules that are introduced by locks.

4.6 Threats to Validity

A threat to the validity of this experimental evaluation is that our subject programs are relatively small; it is difficult to find large open-source RAPs. Large RAPs may have millions of lines of code and use databases whose sizes are measured in thousands of tables and attributes. Those RAPs may have different characteristics compared to our smaller subject programs. On the one hand, increasing the size of RAPs to millions of lines of code is unlikely to affect the time and space demands of our analyses because PERLATO only considers conflicts among concurrently accessed variables, and by the nature of rule-based programming, rules do not share many variables. Thus, the majority of the source code of RAPs is ignored in the conflict analysis, which is focused on concurrent accesses to shared variables among rules. Evaluating this impact is a subject of future work.

Additional threats to validity of this study is that we used graduate students as programmers who created RAPs, and this task should be tackled by professional programmers. However, many of these students have at least one year of professional programming experience, thereby reducing this threat to validity. The other threat to validity is that we tried to avoid complex logics, and we issued instructions to students to restrict the subject RAPs by writing rules under the default agenda group `MAIN` of drools and also by writing rules without many complicated attributes such as `ruleflow-group, activation-group, date-expires`.

Finally, recall that our conflict analysis is conservative and there are many false positives. Improving the precision of this analysis may also improve the overhead of PERLATO. It is also unclear how well PERLATO will perform on many different and diverse RAPs, so this is a threat to external validity of our results.

5. RESULTS

In this section, we report the results of the experiment and evaluate the null hypotheses. We use one-way ANOVA and t-tests for paired two sample for means to evaluate the hypotheses that we stated in Section 4.5. Results of our experiments are provided in Table 2. We used ANOVA to evaluate the null hypothesis H_0 that

Table 2: **PERLATO experimental results. The first four columns specify the name of the RAP, the sequence number for collections of input facts that are inputs for the RAPs, the number of asserted facts and the total number of fired rules during the execution of the RAPs. Next columns report average, median, minimum, and maximum execution times for two types of approaches (DROOLS and PERLATO). Two types of DROOLS approach are (sequential and parallel) and three types of PERLATO approach are rule level, atomic level and variable level for 50 execution runs of each experiment.**

Application	*Input Set*	*Facts*	*Rules Fired*	*Approach*	*Type*	*Avg*	*Med*	*Min*	*Max*	σ^2
EEWS	1	246896	1088	DROOLS	Seq	32.22	32.08	31.559	35.358	0.4
					Par	9.52	9.27	8.267	11.789	0.87
				PERLATO	Rule	13.44	13.6	10.927	14.911	0.99
					Atomic	10.85	10.82	10.21	11.867	0.14
					Variable	10.67	10.67	10.21	11.359	0.06
	2	334220	1454	DROOLS	Seq	42.31	42.12	41.508	43.322	0.21
					Par	11.62	11.53	10.299	14.126	0.85
				PERLATO	Rule	18.86	19.02	15.794	21.549	2.07
					Atomic	15.62	15.58	15.016	16.476	0.11
					Variable	15.08	15.02	14.24	16.689	0.21
	3	247113	572	DROOLS	Seq	19.51	19.44	18.906	21.115	0.24
					Par	6.05	6.19	3.947	8.242	1.32
				PERLATO	Rule	8.33	8.22	7.025	9.442	0.42
					Atomic	7.24	7.13	6.837	8.259	0.13
					Variable	6.79	6.76	6.662	7.165	0.02
TAXC	1	16	95	DROOLS	Seq	20.4	20.32	20.023	20.97	0.06
					Par	19.52	19.52	19.115	20.072	0.07
				PERLATO	Rule	19.71	19.69	19.232	20.657	0.13
					Atomic	19.62	19.61	19.144	20.227	0.08
					Variable	19.55	19.52	19.077	20.161	0.08
	2	22	162	DROOLS	Seq	42.48	42.48	41.553	43.63	0.32
					Par	34.27	34.6	31.866	36.019	1.13
				PERLATO	Rule	36.87	36.83	36.016	37.812	0.19
					Atomic	36.86	36.77	36.007	37.775	0.21
					Variable	36.8	36.74	35.972	37.698	0.23
	3	10	66	DROOLS	Seq	11.78	11.73	11.549	12.344	0.04
					Par	9.85	9.87	9.116	10.314	0.05
				PERLATO	Rule	9.93	9.9	9.641	10.431	0.03
					Atomic	9.91	9.92	9.145	10.342	0.04
					Variable	9.98	9.97	9.309	10.761	0.06
IMS	1	17	124	DROOLS	Seq	8.58	8.59	8.102	8.838	0.01
					Par	2.42	2.42	1.543	3.721	0.11
				PERLATO	Rule	3.08	3.06	2.862	3.618	0.02
					Atomic	3.07	3.08	2.809	3.598	0.02
					Variable	4.46	4.44	4.35	4.736	0.01
	2	75	550	DROOLS	Seq	60.02	60.02	58.883	61.681	0.44
					Par	11.86	11.97	7.074	16.414	4.51
				PERLATO	Rule	10.94	11.05	9.41	12.579	0.42
					Atomic	10.02	10.04	9.242	10.948	0.23
					Variable	15.79	15.78	15.499	16.362	0.01
	3	120	880	DROOLS	Seq	37.76	37.66	37.502	39.536	0.09
					Par	23.29	24.51	15.33	27.849	10.26
				PERLATO	Rule	19.7	19.73	18.43	21.88	0.55
					Atomic	18.78	18.69	17.022	21.016	1.1
					Variable	32.73	32.92	30.445	34.785	0.99

the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the groups for R, A, and V for RAPs as shown in Table 3. Based on these results we can reject the null hypothesis for RAPs EEWS and IMS and we accept the alternative hypothesis H_1. We accept the null hypothesis for the application TAXC. One explanation for TAXC is that the code for rules that concurrently access variables contains very few statements that can be executed in parallel; as a result, R, A, and V synchronization lock approaches do not lead to increased parallelism and subsequently, higher speedup values.

Not surprisingly, the speedup is the highest between the fully unsynchronized parallel execution and fully sequential execution – the maximum speedup is close to 600%, with races, of course. However, an average speedup is 225% across three subject RAPs for all three synchronization approaches. These numbers are in the ballpark of the reported results from a major insurance company whose REMP is fully parallelized and where synchronization was

Table 3: Results of ANOVA tests for sets of input facts.

RAP	*Input*	*F*	F_{crit}	*p*	Test H_0
EEWS	1	77.3	4.4	$\approx 6.2 \cdot 10^{-8}$	Accept
	2	65.6	3.35	$\approx 4.3 \cdot 10^{-11}$	Accept
	3	31.3	3.35	$\approx 9.2 \cdot 10^{-8}$	Accept
TAXC	1	1.97	3.35	0.16	Reject
	2	0.19	3.35	0.83	Reject
	3	2,41	3.35	0.1	Reject
IMS	1	789.6	3.35	$\approx 1.1 \cdot 10^{-24}$	Accept
	2	256.5.6	3.35	$\approx 2.7 \cdot 10^{-18}$	Accept
	3	789.6	3.35	$\approx 1.1 \cdot 10^{-24}$	Accept

Table 4: Results of experiments with symbiotic schedules.

Application	*Input*	*Best Schedule*	*Avg*	*Var*
EEWS	1	10.489	10.4	0.01
	2	15.339	15.19	0.06
	3	6.923	6.95	0.01
TAXC	1	19.428	19.57	0.05
	2	36.028	36.8	0.06
	3	9.286	9.55	0.09
IMS	1	3.084	2.76	0.01
	2	9.297	9.43	0.06
	3	17.409	17.32	0.82

experimentally tried at our request. Interestingly, the company suggested that they may go with a higher speedup and tolerate incon.

To test the null hypothesis H1, H2, and H3 we applied two t-tests for paired two sample for means, for elapsed execution times for different experiments. Based on these results we reject the null hypotheses H1 and H3, and we accept the alternative hypotheses that say that **atomic-level synchronizations result in higher speedups than using rule and variable-level synchronization locks**. The hypothesis H2 is accepted, leading us to conclude that **using variable-level synchronization locks does not lead to higher speedups when compared with rule-level synchronization locks**. Possible explanations include a significantly increased overhead of locking on the variable level and greater difficulty to find an optimal schedules for rule executions.

The results of experiments for computing an optimal symbiotic schedule are shown in Table 4 for 10 independent runs of subject RAPs with schedules that are randomly permuted. These results suggest that it is possible to identify an optimal schedule among rules that concurrently access resources that are protected by using synchronization objects. Obtaining sample runs for computing a symbiotic schedule with different permuted schedules can be done during testing of RAPs. Of course, more research is needed to investigate this issue, and we report preliminary results in this paper.

Summary. Based on our experimental results, we can answer affirmatively to RQ1 that PERLATO is effective in increasing speedup for subject RAPs and to RQ3 that symbiotic scheduling is effective in obtaining higher speedup for RAPs. However, our answer is negative to RQ2, since we determined that a finer granularity locking strategy is not more effective in speeding up RAPs.

6. RELATED WORK

Related work to PERLATO consists of two sections: research on detecting and eliminating races in multithreaded applications and research for improving reliability of rule-driven applications. We believe that our work is the first at the intersection of these two important areas.

There is a multitude of research in detecting and preventing races in multithreaded applications and we concentrate on related work that goes beyond race detection in multithreaded Java and C++ applications. There is an excellent survey for race detection and prevention techniques [9], however, it does not discuss RAPs. Neither does a paper on taxonomy of race conditions mentions REMPs [37]. Our ideas of fast and large-scale race detection are related to RacerX [20], however, it is unclear how RacerX can be applied to RAPs. Recent work includes partial detection and prevention of certain races in Ajax applications [1], file systems [73], workflow applications [72], relational databases [28, 40], and web services [79], however, no work is done in rule-driven applications. CARISMA is a recent work that is related to PERLATO, where dynamic race detectors explore multiple thread schedules of a multithreaded program over the same input to detect data races [78]. In contrast, PERLATO uses a combination of static and dynamic analysis among rules in RAPs, however, ideas of CARISMA are complementary to PERLATO. An interesting use of thread scheduling for executing multiple replicas of the program may lead to prevention of data races, and we may research the application of this approach to RAPs in future work [74].

The contribution of the software engineering community to rule-based programming is limited. Weyuker et al published one of first papers where an algorithm is presented for reliability testing of rule-based systems [3]. Same authors published a paper eight years later on estimating CPU utilization in rule-based systems [4]. Some approaches on testing knowledge systems use the ideas of rule-based development [54, 44, 34, 6], however, we know of no papers that investigated races in RAPs using the fundamental notion of separation of concern like we do in PERLATO. A related area of research investigates checking consistency and completeness of rule-based expert systems [57, 50, 19]. An idea of using control and data flow in testing RAPs was proposed by Barr [7], but it was never applied to deal with races. Some approaches deal with error checking and bug finding in RAPs[66]. However, these approaches do not address races in RAPs.

7. CONCLUSION

RulE Management Platforms (REMPs) are widely used in enterprise applications in which programming logic is represented using *rules*, which are executed sequentially by REMPs. We created a novel solution that is based on obtaining a rule execution model that is used at different layers of REMPs to enhance the performance of rule-driven applications while maintaining their reliability and adaptability. First, using this model, possible races are detected statically among rules, and we evaluate an implementation of our abstraction of algorithms for automatically preventing races among rules. Next, we use the sensitivity analysis to find better schedules among simultaneously executing rules to improve the overall performance of the application. We implemented our solution for JBoss Drools and we evaluated it on three applications. The results suggest that our solution is effective, since we achieved over 225% speedup on average.

Acknowledgments

This work is supported by NSF CCF-1217928, CCF-1017633, and Microsoft SEIF. We warmly thank Chen Fu and Andrea Bonisiol for their contributions at the initial stage of the project.

8. REFERENCES

[1] T. J. Albert, K. Qian, and X. Fu. Race condition in ajax-based web application. ACM-SE 46, pages 390–393, New York, NY, USA, 2008. ACM.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Softw.*, 13(5):76–82, Sept. 1996.

[4] A. Avritzer, J. P. Ros, and E. J. Weyuker. Estimating the cpu utilization of a rule-based system. WOSP '04, pages 1–12, New York, NY, USA, 2004. ACM.

[5] M. Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.

[6] V. Barr. Applications of rule-base coverage measures to expert system evaluation. In *Journal of Knowledge Based Systems*, pages 411–416. Press/ MIT Press, 1998.

[7] V. Barr and D. V. Barr. Rule-based system testing with control and data flow techniques, 1996.

[8] D. S. Batory, R. Gonc_c alves, B. Marker, and J. Siegmund. Dark knowledge and graph grammars in automated software design. In *SLE*, pages 1–18, 2013.

[9] N. E. Beckman. A survey of methods for preventing race conditions, 2006.

[10] A. Ben-David. Rule effectiveness in rule-based systems: A credit scoring case study. *Expert Syst. Appl.*, 34(4):2783–2788, May 2008.

[11] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<t>: A first-order type for uncertain data. In *ASPLOS*, pages 239–248, 2014.

[12] E. A. Brewer. Towards robust distributed systems (abstract). PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[13] P. Browne. *JBoss Drools Business Rules*. Packt Publishing, 2009.

[14] B. G. Buchanan and R. O. Duda. Principles of rule-based expert systems. Technical report, Stanford University, Stanford, CA, USA, 1982.

[15] S. Carr, J. Mayo, and C.-K. Shene. Race conditions: a case study. *J. Comput. Sci. Coll.*, 17(1):90–105, Oct. 2001.

[16] L. Chung, K. Cooper, and A. Yi. Developing adaptable software architectures using design patterns: An nfr approach. *Comput. Stand. Interfaces*, 25(3):253–260, June 2003.

[17] R. Dazeley, P. Warner, S. Johnson, and P. Vamplew. The ballarat incremental knowledge engine. In *Proceedings of the 11th PKAW*, PKAW'10, pages 195–207, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[19] R. Djelouah, B. Duval, and S. Loiseau. Validation and reparation of knowledge bases. In *Proceedings of the 13th ISMIS '02*, ISMIS '02, pages 312–320, London, UK, UK, 2002. Springer-Verlag.

[20] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, pages 301–312, 2012.

[22] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, pages 449–460, 2012.

[23] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. *IEEE Micro*, 33(3):16–27, 2013.

[24] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *ASPLOS*, pages 91–102, 2010.

[25] M. Fayad and M. P. Cline. Aspects of software adaptability. *Commun. ACM*, 39(10):58–59, Oct. 1996.

[26] M. Fowler. Should i use a rules engine? *martinfowler.com*, Jan. 2009.

[27] H. Gaur and M. Zirn. *Oracle Fusion Middleware Patterns*. Packt Publishing, 2010.

[28] S. Ghandeharizadeh and J. Yap. Gumball: a race condition prevention technique for cache augmented sql database management systems. DBSocial '12, pages 1–6, New York, NY, USA, 2012. ACM.

[29] J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.

[30] J. C. Giarratano and G. D. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2005.

[31] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *ESEM*, 2010.

[32] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proceedings of ISCA*, ISCA '86, pages 28–37, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[33] K. Harris-Ferrante and S. Forte. Hype cycle for p&c insurance. *Gartner*, July 2009.

[34] R. Hartung and A. Hρ akansson. Automated testing for knowledge based systems. KES'07/WIRN'07, pages 270–278, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] D. Heckerman and E. Horvitz. The myth of modularity in rule-based systems. *CoRR*, abs/1304.3090, 2013.

[36] J. Hedberg, K. Weare, and M. la Cour. *MCTS: Microsoft BizTalk Server 2010 (70-595) Certification Guide*. Packt Publishing, 2012.

[37] D. P. Helmbold and C. E. McDowell. A taxonomy of race conditions. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1994.

[38] S. D. Hendrick. Worldwide business rules management systems 2009-Ű2013 forecast update and 2008 vendor shares. *IDC*, Oct. 2009.

[39] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[40] J. M. Hughes and H. Bolinder. Testing a database for race conditions with quickcheck: none. Erlang '11, pages 72–77, New York, NY, USA, 2011. ACM.

[41] P. Jackson. *Introduction to Expert Systems, 3rd Edition*. Addison-Wesley, 1999.

[42] R. J. K. Jacob and J. N. Froscher. A software engineering methodology for rule-based systems. *IEEE Trans. on Knowl. and Data Eng.*, 2(2):173–189, June 1990.

[43] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[44] J. D. Kiper. Structural testing of rule-based expert systems. *ACM Trans. Softw. Eng. Methodol.*, 1(2):168–187, Apr. 1992.

[45] L. Lahav. Hobbes framework: An adaptable solution to web-driven applications. *Comput. Stand. Interfaces*, 25(3):271–274, June 2003.

[46] B. A. Lieberman. Requirements for rule engines. *IBM DeveloperWorks*, Nov. 2012.

[47] L. Lin, S. M. Embury, and B. C. Warboys. Facilitating the implementation and evolution of business rules. ICSM '05, pages 609–612, Washington, DC, USA, 2005. IEEE Computer Society.

[48] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, Mar. 2008.

[49] S. Lu, S. Park, and Y. Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Trans. Softw. Eng.*, 38(4):844–860, July 2012.

[50] S. Lukichev. Improving the quality of rule-based applications using the declarative verification approach. *Int. J. Knowl. Eng. Data Min.*, 1(3):254–272, Dec. 2011.

[51] S. Luypaert. Rule engines in java: Jboss drools. *Java, JBoss Drools*, July 2010.

[52] A. Mccallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *In Advances in Neural Information Processing Systems 22*, pages 1249–1257, 2009.

[53] D. W. McCoy. Taking the mystery out of business rule representation. *Gartner*, Mar. 2009.

[54] T. Menzies and B. Cukic. On the sufficiency of limited testing for knowledge based systems. ICTAI '99, pages 431–, Washington, DC, USA, 1999. IEEE Computer Society.

[55] C. Moran. Does your project need a rule engine? *Java Developer's Journal*, June 2004.

[56] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[57] T. A. Nguyen, W. A. Perkins, T. J. Laffey, and D. Pecora. Checking an expert systems knowledge base for consistency and completeness. IJCAI'85, pages 375–378, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[58] Oracle. Oracle fusion middleware user's guide for oracle business rules. *Oracle Documentation,*`http://docs.oracle.com/cd/E21764_01/integration.1111/e10228/intro.htm`, May 2011.

[59] S. Purohit and K. Jamdaade. Rule based system to facilitate the immunity of hiv/aids patients using ayurveda therapy. CUBE '12, pages 226–234, New York, NY, USA, 2012. ACM.

[60] redhat. Why use a rule engine? *Customer Portal,*`https://access.redhat.com/site/documentation/en-US/JBoss_Enterprise_SOA_Platform/4.2/html/JBoss_Rules_Manual/sect-JBoss_Rules_Reference_Manual-Why_use_a_Rule_Engine.html`, Feb. 2013.

[61] D. Roy. Probabilistic-programming.org. `http://probabilistic-programming.org/wiki/Home`, Feb. 2014.

[62] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer*. Wiley-Interscience, Hoboken, NJ, Feb. 2008.

[63] W. Schulte and J. Sinur. Rule engines and event processing. *Gartner*, Mar. 2009.

[64] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[65] J. Sinur. The art and science of rules vs. process flows. *Gartner*, Mar. 2009.

[66] C. Sinz, T. Lumpp, J. Schneider, and W. Küchlin. Detection of dynamic execution errors in {IBM} system automation's rule-based expert system. *Information and Software Technology*, 44(14):857 – 873, 2002.

[67] S. Smith and A. Kandel. *Verification and Validation of Rule-Based Expert Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1994.

[68] A. Snavely, D. M. Tullsen, and G. M. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, pages 66–76, 2002.

[69] stackoverflow. Rules engine - pros and cons. *stackexchange,*`http://stackoverflow.com/questions/250403/rules-engine-pros-and-cons/398389#398389`, Dec. 2008.

[70] stackoverflow. When should you not use a rules engine? *stackexchange,*`http://stackoverflow.com/questions/775170/when-should-you-not-use-a-rules-engine`, Nov. 2011.

[71] N. Subramanian and L. Chung. Software architecture adaptability: An nfr approach. IWPSE '01, pages 52–61, New York, NY, USA, 2001. ACM.

[72] X. Sun, A. Agarwal, and T. S. E. Ng. Attendre: mitigating ill effects of race conditions in openflow via queueing mechanism. ANCS '12, pages 137–138, New York, NY, USA, 2012. ACM.

[73] P. Uppuluri, U. Joshi, and A. Ray. Preventing race condition attacks on file-systems. SAC '05, pages 346–353, New York, NY, USA, 2005. ACM.

[74] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. SOSP '11, pages 369–384, New York, NY, USA, 2011. ACM.

[75] C.-C. Wu. Parallelizing a clips-based course timetabling expert system. *Expert Syst. Appl.*, 38(6):7517–7525, June 2011.

[76] R. M. Wygant. Clips - a powerful development and delivery expert system tool. *Comput. Ind. Eng.*, 17(1):546–549, Nov. 1989.

[77] M. G. Yunusoglu and H. Selim. A fuzzy rule based expert system for stock evaluation and portfolio construction: An application to istanbul stock exchange. *Expert Syst. Appl.*, 40(3):908–920, Feb. 2013.

[78] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse. Carisma: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. ISSTA 2012, pages 221–231, New York, NY, USA, 2012. ACM.

[79] J. Zhang, S. Su, and F. Yang. Detecting race conditions in web services. AICT-ICIW '06, pages 184–, Washington, DC, USA, 2006. IEEE Computer Society.