2009

Regression Model Checking

Guowei Yang
gyang@cse.unl.edu

Matthew B. Dwyer
dwyer@cse.unl.edu

Gregg Rothermel
University of Nebraska-Lincoln, grothermel2@unl.edu

IEEE International Conference on Software Maintenance, 2009. ICSM 2009.
Digital Object Identifier: 10.1109/ICSM.2009.5306334

Regression Model Checking

Guowei Yang, Matthew B. Dwyer, Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{gyang,dwyer,grother}@cse.unl.edu

Abstract

Model checking is a promising technique for verifying program behavior and is increasingly finding usage in industry. To date, however, researchers have primarily considered model checking of single versions of programs. It is well understood that model checking can be very expensive for large, complex programs. Thus, simply reapplying model checking techniques on subsequent versions of programs as they evolve, in the limited time that is typically available for validating new releases, presents challenges. To address these challenges, we have developed a new technique for regression model checking *(RMC), that applies model checking incrementally to new versions of systems. We report results of an empirical study examining the effectiveness of our technique; our results show that it is significantly faster than traditional model checking.*

1. Introduction

Model checking is a promising technique for verifying that programs are free of, and detecting subtle instances of, certain types of errors. Notable success has been achieved, for example, in analyzing and detecting classes of errors in the Linux kernel [19], TCP/IP implementations [14], and widely-used file system implementations [22].

Despite such success, it is also well understood that model checking can be expensive. Thus, a large body of research has focused on developing techniques for reducing analysis cost through property preserving state-space reductions (e.g., [10]), and abstraction techniques (e.g., [1]). These techniques have greatly increased the size and complexity of systems that can be model-checked, yet model checking remains among the most costly of software validation and verification techniques.

Verifying properties of software systems is important, but systems that succeed, evolve, and engineers must revalidate new system versions to ensure that they have not regressed, and that new functionality works as intended. This need has led to many new approaches in the realm of regression testing (e.g., [13, 18]); however, to date, most research on model checking has considered its application only to single versions of programs.

The application of model checking in evolutionary contexts presents many challenges. Model checking systematically explores all reachable states of a system, and the expense of doing this means that simply reapplying full model checking techniques to programs as they evolve, in the limited time that is available for validating new releases, may be infeasible. Reapplying model checking incrementally may reduce this expense. Such reapplication, however, will require model checking techniques to collect and use analysis data from prior releases in a cost-effective manner.

To address these challenges, we have developed a new technique for *regression model checking* (RMC), that applies model checking incrementally to new versions of systems. RMC reuses data obtained through model checking of earlier versions, leveraging the facts that state spaces of consecutive versions tend to be similar and the influence of changes tends to be localized within regions of a state space.

More specifically, RMC can be run in a *recording* mode where it calculates and stores, for each program state, a set of program coverage elements (e.g., basic blocks) that are reached by program executions that continue from the state. When a program change is made, impact analysis is used to calculate the coverage elements whose behavior may now differ; these are referred to as *dangerous elements*. RMC can then be run in a *pruning* mode where it checks whether the reachable elements recorded for the state include any dangerous elements. If no dangerous elements are reachable, then the portion of the state-space that is rooted at the given state is guaranteed to be the same as during the recording run of RMC; we refer to this as a *safe* sub-state space. RMC can skip exploration of any safe sub-state space.

Our RMC technique is orthogonal to any reduction, abstraction, or bounding techniques applied during model checking. RMC preserves property checking relative to a non-regression model check: properties that hold in one will hold in the other and violations detected by one will be reported by the other. Thus, RMC is lossless relative to the underlying model checking approach.

978-1-4244-4828-9/09/$25.00 2009 IEEE

We have implemented our RMC technique in Java PathFinder (JPF), but the technique is applicable in any explicit-state model checking setting. We report the results of an empirical study examining our technique's effectiveness on a variety of programs and versioning changes. Our results show that RMC is significantly faster than traditional model checking, a result that should facilitate the use of model checking on programs as they evolve.

2. Overview

A Basic Process Model. In this work, we assume a model of the maintenance and verification process that corresponds to the most commonly used approach for validating evolving systems [16] — a *batch* process model — which, though simple, is sufficient to let us investigate RMC.

Figure 1 presents a timeline depicting the maintenance, verification, and post-release phases for the production of a new release of an evolving software system under this model. Time $t1$ represents the time, following completion and shipment of a prior release, at which maintenance (enhancements and corrections) directed at the new release begins. At time $t2$ the new release is code-complete, and verification and fault correction begin. When this phase ends, at time $t3$, product release occurs; at this time, revenue and other benefits can begin to accrue. Typically, a new maintenance phase also begins during this post-release interval.

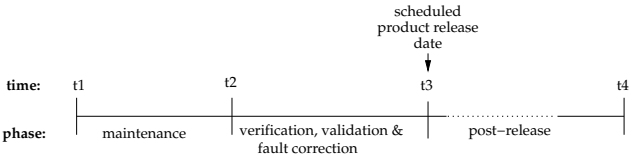

Figure 1. Maintenance & verification lifecycle

In this batch process model, time interval $t2 - t3$ is the *critical period* [18] for validation activities; it is usually time-limited, and its extent determines whether product release occurs on time, with affects on revenue. Time interval $t1 - t2$, on the other hand, is the *preliminary period*, usually much longer than the critical period. Validation activities can take advantage of the preliminary period to gather data and prepare for time-constrained critical period activities.

In this paper, where our interest is model checking, we need to reduce the effort required to model check a new system release during the critical period. A full model check of the system on every release may be inordinately expensive, but our thesis is that an incremental approach that leverages data gathered in the preliminary period to reduce critical period costs will be feasible, and will enable the application of model checking to evolving systems, enhancing reliability.

A Two-phase State Space Search. We exploit the process described above by developing specialized variants of the basic model checking algorithm (i.e., depth-first search (DFS) of the program's state space.)

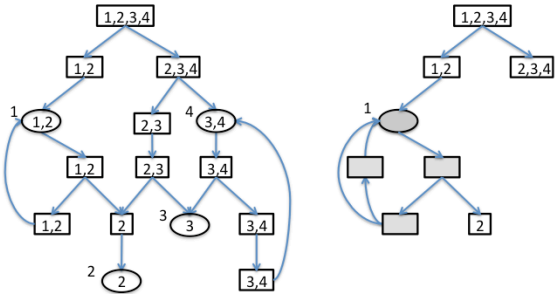

Figure 2. Recording and pruned state spaces

The first phase of our approach traverses the program state space just as a traditional model checker would. We piggyback onto this traversal the calculation and recording of program coverage elements for each program state. The left side of Figure 2 illustrates the results of this recording phase on a hypothetical example. This example has 15 states, depicted as ovals and rectangles, with directed edges representing state transitions. In this example, we use a partial mapping from states to coverage elements (e.g., the first statement in a method body) and thus only four of the states (depicted as ovals) map to coverage elements; the element number for those states is shown to the upper left of each oval. Each state is labeled with the set of coverage elements that can be reached by some sequence of transitions; for example, the leftmost oval is labeled by $1, 2$ since it maps to coverage element 1 and through a sequence of three transitions (down and to the right) the state which maps to coverage element 2 can be reached. The *reachable elements* for a state are calculated during the DFS by accumulating reachable elements during DFS backtracking; these are stored, along with the states, for later use.

Imagine a change to the program that impacts only coverage element 1; element 1 is the only *dangerous element*. The second phase exploits this information to *prune* portions of the DFS for the changed program. In particular, when the search encounters a state that appeared in the recording phase's state space and for which none of its reachable elements are dangerous the DFS can immediately backtrack. The right side of Figure 2 illustrates this pruning for the states labeled "2,3,4" and "2" which saves the cost of exploring eight states. The program change depicted here changes the state space; in this case the shaded states are changed with a new state added that provides an alternative cycle back to the state labeled with element 1. Note however that any impacted portion of the state space, including any new states, is guaranteed to be searched.

Our two-phase RMC approach trades time and space during the preliminary period of maintenance to reduce the cost of analysis during the critical period. The results of our empirical study, in Section 5, demonstrate that significant savings can be achieved during the critical period by using this approach. Moreover, for four of the five programs that

we studied, the time required to run *both* phases of RMC was actually less than the time required to run a traditional full model check twice.

While the space overhead of our RMC approach can be significant, we believe that there are many strategies for reducing it (see Section 6) and that trading space for reduction in runtime is consistent with memory technology trends.

3. Background

Regression Testing. Our regression model checking approach makes use of data gathered during the application of a regression test selection technique. Let P be a program, let P' be a modified version of P, and let T be a test suite developed for P. Regression testing is concerned with validating P'. Reusing all of T can be expensive, so *regression test selection* (RTS) techniques (see [17]) use data on P, P', and T to select a subset T' of T with which to test P'. One class of RTS techniques, *safe* techniques, (e.g. [18]) guarantee that under certain conditions, test cases not selected could not have exposed faults in P' [17]. Empirical studies have shown that these techniques can be cost-effective.

In this work we use a safe RTS technique, `Dejavu` [18], to provide data for RMC. `Dejavu` constructs control-flow graph (CFG) representations of the procedures in P and P', in which individual nodes are labeled by their corresponding statements. `Dejavu` utilizes test trace information that records, for each test case t in T and each edge e in the CFG for P, whether t traversed e. Given this information, `Dejavu` performs a simultaneous depth-first graph walk on a pair of CFGs G and G' for each procedure and its modified version in P and P', following identically-labeled edges, to find code changes. Given two edges e and e' in G and G', if the code associated with nodes reached by e and e' differs, we call e a *dangerous edge*: it leads to code that may cause program executions to exhibit different behavior.

In the regression test selection context, encountering a dangerous edge is occasion for selecting tests, from T, that are known to reach that edge in P. In this work, however, it is the identification of dangerous edges that interests us, as these provide the information needed to drive RMC.

Model Checking. Program model checking, of the form implemented in JPF [21], views a program as a guarded-transition system and analyzes transition sequences to infer properties of program executions. A *guarded transition system* consists of a set of variables, which for our purposes are coalesced into a single composite *state variable* s, and a set of guarded transitions which atomically test, with predicate ϕ, the current state and update the state by executing a transition, α, i.e., if $\phi(s)$ then $s = \alpha(s)$. The initial values of program variables are used to define an initial state, s_0.

Figure 3 presents the basic DFS algorithm that generates the program state-space terminating when it finds an error or all reachable states. The `basicDFS` algorithm initializes

```
basicDFS()                          DFS(s)
 1 seen := {s0}                      4 for α ∈ enabled(s) do
 2 push(stack, s0)                   5   s' := α(s)
 3 DFS(s0)                           6   if error(s') then
end basicDFS()                       7     ce := stack
                                     8     exit
                                     9   if s' ∉ seen then
                                    10     seen := seen ∪ {s'}
                                    11     push(stack, s')
                                    12     DFS(s')
                                    13     pop(stack)
                                    end DFS()
```

Figure 3. Basic DFS

the set of states *seen* in the search and the *stack* that stores the current path in the state-space being analyzed, and then starts a recursive DFS from the initial state. Lines 4-13 comprise a step in the DFS search. On line 4, *enabled*(s) returns the set of transitions, α, whose guard, ϕ, is true in the given state. The loop iterates through the set of enabled transitions in a fixed order considering each successor state. Lines 6-8 test whether an error state has been reached, and, if so, record the current DFS stack as a counterexample before terminating the search. Lines 9-13 test whether the successor state has been seen previously and if not launch a recursive search from that state.

4. Regression Model Checking

Traditional model checking attempts to generate all states in the space of program behaviors in order to prove the absence of, or detect the presence of, errors. Many techniques have been explored for reducing the effort required to perform a model check. Some techniques (e.g., [1, 10]) reduce effort while preserving error detection, while other techniques, for example, bound the number of transitions explored along a path, reducing effort while sacrificing error detection.

The main idea behind RMC is to reduce effort and preserve error detection relative to the application of non-regression model checking applied to a new release of software during the critical period. RMC achieves this by avoiding checking some safe sub-state spaces given the data gathered in the preliminary period.

RMC works by following three steps: (1) a recording phase of RMC (recordingRMC) gathers data in model checking an earlier version, (2) `Dejavu` computes the dangerous edges with respect to the earlier and new version of the program, and (3) a pruning phase of RMC (pruningRMC) determines which sub-state spaces in the new version are safe and prunes them.

4.1. Algorithms

Figure 4 shows the RMC algorithms: `recordingRMC` and `pruningRMC`. These algorithms are variations on the DFS algorithm shown in Figure 3. To highlight the differences between the algorithms we label steps that are added,

relative to the basic DFS, with letter suffixes; the labels of all modified steps are shown in a bold font.

The `recordingRMC` algorithm initializes the reachable elements, counterexamples maps and an error flag (lines 0a-0b) that are used to record information needed for the pruning model check. On line 4a any coverage elements associated with the state transition, $e(\alpha)$, are added to the reachable elements from the current state, $r(s)$. Unlike the basic DFS algorithm, DFS_r returns a set of reachable elements encountered in states reached later in the search. Line 12 accumulates the elements found when a successor has not been previously encountered in the search, whereas lines 13c-13i accumulate elements that were previously calculated for a successor state that was already seen. Lines 13d-13g deal with cycles in the state space by iterating down the stack unifying the *r* sets for all the states in the current cycle, and forcing those states to subsequently share (denoted by $\equiv$) a single *r* set. The counterexample map records whether there is an error in a successor state of a given state by storing a counterexample(s) for those errors. When an error is found, lines 7a-7b update the map with the current counterexample, i.e., *stack*, and lines 8a-8b end the search while updating, lines 12 and 13a, the reachable elements for all states on the *stack*.

The `pruningRMC` algorithm takes as input dangerous edges computed by `Dejavu` and information computed by `recordingRMC`. If there are no dangerous edges (line 0a) then the new release is equivalent to the the earlier release. If a counterexample was found in the previous version, it is reported again in this case (lines 0b-0d). Lines 3a-3g test whether any dangerous edge is reachable from the current state. If not, then the sub-state space rooted at the current state is safe and can be pruned. If a counterexample was found in the safe sub-state space in the previous version it is reported (lines 3c-3e); otherwise the DFS returns. This mechanism reduces the effort required to model check the new release, and guarantees that RMC is as effective as full model checking in terms of error finding capability.

Clearly `recordingRMC` explores the same state space as a non-regression model check. `pruningRMC` preserves results for state properties, e.g., assertions or invariant checks, since it skips only states that are guaranteed to be identical to those found during `recordingRMC`, and property violations for those states are recorded and reported. In future work, we plan to explore the recording of Buchi automaton states along with reachable elements to preserve LTL property checking results.

4.2. Implementation

We implemented RMC in JPF, an explicit state model checker for Java programs. JPF executes Java bytecode programs; it is a Java Virtual Machine (JVM) running on top of a host JVM to verify all states of the checked programs.

```
r "Reachable Elements Map"
c "Counterexamples Map"

recordingRMC()
 0a  r, c := new, new
 0b  err := false
 1   seen := {s0}
 2   push(stack, s0)
 3   DFS_r(s0)
end recordingRMC()

DFS_r(s)
 4  for α ∈ enabled(s) do
 4a    r(s) := r(s) ∪ e(α)
 5     s' := α(s)
 6     if error(s') then
 7       ce := stack
 7a      for s_e ∈ stack do
 7b        c(s_e) := ce
 8a      err := true
 8b      return r(s)
 9     if s' ∉ seen then
 10      seen := seen ∪ {s'}
 11      push(stack, s')
 12      r(s) := r(s) ∪ DFS_r(s')
 13      pop(stack)
 13a     if err = true then break
 13b   else
 13c     if s' ∈ stack then
 13d       for i ∈ stack.top... do
 13e         if stack[i] = s' then break
 13f         r(s') := r(s') ∪ r(stack[i])
 13g         r(stack[i]) ≡ r(s')
 13h     else
 13i       r(s) := r(s) ∪ r(s')
 14  return r(s)
end DFS_r()
```

```
d "Dangerous Edges"

pruningRMC()
 0a if d = ∅ then
 0b   if c(s0) ≠ ∅ then
 0c     ce := c(s0)
 0d     exit
 0e   return
 1   seen := {s0}
 2   push(stack, s0)
 3   DFS_p(s0)
end pruningRMC()

DFS_p(s)
 3a if r(s) ≠ ∅ then
 3b   if d ∩ r(s) = ∅ then
 3c     if c(s) ≠ ∅ then
 3d       ce := c(s)
 3e       exit
 3f     else
 3g       return
 4  for α ∈ enabled(s) do
 5    s' := α(s)
 6    if error(s') then
 7      ce := stack
 8      exit
 9    if s' ∉ seen then
 10     seen := seen ∪ {s'}
 11     push(stack, s')
 12     DFS_p(s')
 13     pop(stack)
end DFS_p()
```

Figure 4. RMC algorithms

JPF is an open source system with support for extension. It provides listeners, `SearchListener` and `VMListener`, to facilitate customization. We implemented two listeners: `recordingRMCListener` and `pruningRMCListener`. The former computes a reachable elements map during JPF verification, and the latter loads the stored reachable elements map and dangerous elements information for the new release and controls the verification to prune states by backtracking during the search.

`Dejavu` is available in Sofya [11], which is a framework providing program analysis capabilities to facilitate the testing, maintenance and optimization of Java code.

To facilitate the recognition of the reachable elements in JPF, we instrument the program bytecodes using BCEL [2]. Unique identifiers are defined for each coverage element of interest (e.g. blocks) in the program; these identifiers are expressed in the format used by `Dejavu`. The instrumentation marks each element in the program with its identifier. A `VMListener` extends JPF so that the execution of instrumented bytecodes is recognized, the element identifier is extracted, and it is transfered to the RMC algorithm; this is how $e(\alpha)$ in line 4a of DFS_r is implemented.

Our implementation adopts similar data structure choices to those in the JPF code base. While we have not focused on optimizing RMC performance, in Section 6 we discuss approaches that might be explored to significantly reduce memory consumption.

5. Empirical Study

Our RMC technique is intended to enable model checking of evolving software by lowering the cost of applying it in comparison to the cost of applying full model checking (FMC) while preserving effectiveness. The preservation of effectiveness by our technique is guaranteed by construction, as discussed in Section 4.[1] Thus, we examine the cost of RMC relative to the cost of FMC. This leads to the following research question:

RQ: How do the costs of applying RMC and FMC to evolving software compare.

We designed an experiment to address this question. This section describes our objects of analysis, variables and measures, experiment setup, threats to validity, and results.

5.1. Objects of Analysis

We used five Java programs as objects of analysis (see Table 1): `Daisy`, `Elevator`, `AlarmClock` (version ac9), `RaxExtended`, and `ReplicatedWorkers`. For each of these objects, the table provides information on its associated "Reference" (the source of the object), "Parameters" (parameters required for the object to run), "Error" (type of error to be verified), "Threads" (number of threads), "Classes" (number of class files), "Methods" (number of methods), "SLOC" (number of lines of code), and "Versions" (number of versions we used in the study).

The programs were selected from the artifacts described in [4]. We restricted our selection to programs classified there as "realistic". The programs perform computations over a rich set of data structures, and most have been previously used for evaluation of state-space search techniques.

To conduct our study we required multiple versions of our objects. Given that multiple versions were not available, we needed to construct versions somehow. We considered two approaches for doing this.

The first approach is to make versions by hand. This has the advantage of being realistic to some extent; however, it also has disadvantages. First, creating a number of versions sufficient to allow statistically significant data sets to be obtained in our studies would be difficult, threatening our power to draw conclusions. Second, our results would be dependent on the specific set of changes made, with potential affects on replicability and generalizability.

[1] As a check on our implementation, however, we did confirm RMC's property preservation in the course of our empirical study by verifying that the errors reported by FMC and RMC were the same.

The second approach we considered is to simulate changes. For example, changes can be simulated by adding "mutations" that randomly change program state. The disadvantage of this approach is that the changes are not real code modifications. An advantage, however, is that it lets us observe effects across a broad range of changes within a large number of modules in our systems, increasing our ability to draw conclusions about the relative costs of the algorithms without the biases inherent in a smaller set of hand-seeded changes. Because our primary goal is to conduct a controlled experiment enabling us to draw such conclusions, we chose this second approach.

The basic idea behind our approach is to simulate two classes of program changes that could be encountered in practice, non-state-changing and state-changing, and to do this at the level of individual methods; that is, simulate the case where, somewhere within the method, one of these classes of changes has occurred. To do this we inserted new initial statements in target methods, that use a random number generator to determine whether to alter the value of a variable. This has two effects: (1) it ensures that `Dejavu` will flag a dangerous edge in the changed method, and (2) it ensures that on some (randomly selected) runs of RMC on the changed version, program state will also change.

For each of our object programs v_{old}, when we apply each simulated change we do so only in one method, thus obtaining one new version v_{new} of that program. Application sites (target methods) were selected randomly with some restrictions. For Daisy and Elevator, given the expense of simulating changes in all methods, we restricted our attention to methods in the classes `DaisyDir` and `Elevator`, which provide the core functionality of these objects and are thus utilized, to some extent, in any execution. For `AlarmClock`, `RaxExtended` and `ReplicatedWorkers`, we excluded library methods that are not executed in the context of the objects, and we excluded methods that simply invoke or serve as interfaces to methods that provide functionality (main() methods, constructor methods, methods in interfaces, and run() methods in Thread). Table 1 indicates the numbers of versions used.

5.2. Variables and Measures

5.2.1 Independent Variables

Our independent variable is the model checking algorithm utilized. We use our RMC algorithms, described in Section 4, and as a control representing currently available practice we use the application of full model checking (FMC), as implemented by JPF in its default mode where JPF finds the first error then exits as described in Section 3.

5.2.2 Dependent Variables and Measures

We chose three dependent variables and measures, all ultimately related to the costs of RMC and FMC. Note that the

Subject	Reference	Parameters	Error	Threads	Classes	Methods	SLOC	Versions
Daisy	[6]	none	AssertionViolation	3	21	106	744	13
Elevator	[15]	none	ArrayIdxOOBExcpn	4	12	96	934	29
AlarmClock	[3]	none	NullPtrExcpn	3	6	20	125	13
RaxExtended	[7]	gc, wc, envFirst	AssertionViolation	6	11	23	127	10
ReplicatedWorkers	[3]	#workers, #items, min, max, epsilon	Deadlock	6	14	50	304	28

Table 1. Object Programs and Associated Data

ways in which costs are measured varies with the technique being applied. Given an original version of an object program v_{old}, which has been transformed into a new version v_{new}, FMC is applied only to v_{new}. Using RMC, recordingRMC is applied to v_{old} and pruningRMC is applied to v_{new}. However, analyses applied to v_{old} are preliminary phase costs (as noted in Section 2) and our primary interest here is critical phase costs. Our formal measures thus focus on those costs. Later, however, in our discussion of results (Section 6) we do comment on preliminary phase costs.

State Space Cost. A primary driver of model checking cost is state space size: JPF checks each state to verify the correctness of a program, during which JPF also needs to compute the hashcode of the state. Thus, the number of states that must be considered determines the cost of model checking to a large extent, and is often used as a measure for the performance of Java state-space search techniques in the literature (e.g., [4]). To track this cost, we measure the number of states visited by FMC on v_{new} and the number of states visited by pruningRMC on v_{new}, and denote these as SS_{FMC} and SS_{RMC}, respectively.

Execution Time. Although the size of the state space visited largely determines the cost of model checking, we also need to measure the execution time for model checking, given that our search listener performs additional checks that may affect performance. We denote the execution time for FMC (on v_{new}) as ET_{FMC} and the execution time for pruningRMC (on v_{new}) as ET_{RMC}.

Memory Usage. To reduce the effort required to model check an evolving system, RMC needs to store and retrieve extra data beyond that needed by FMC. Thus, memory usage is a third cost we consider in our study. We denote the memory usage for FMC (on v_{new}) as MU_{FMC} and the memory usage for pruningRMC (on v_{new}) as MU_{RMC}.

We gather all of the foregoing measures — numbers of states, execution times and memory usage data — from the output generated by the JPF reporting system.

5.3. Experiment Setup

To perform our study, we compiled all object program versions using Java version 1.5.0 and then model checked each version using JPF version 4 with partial order reduction enabled. The study was performed on a Opteron 250 running at 2.4GHz with 16 GByte of memory and running Fedora Core 3 Linux. We set an execution time bound of 2 hours and a memory bound of 8GB, and terminated runs at either of them.

When applying model checking to our object programs, on `Elevator` and `RaxExtended` we used a depth bound. Originally, we proceeded without a depth bound on these two programs, but more than half of the `Elevator` versions and almost all of the `RaxExtended` versions could not run to completion. However, on the limited number of completed cases, we found clear decreases in terms of the number of new states visited and the time cost (not shown in this paper). To give an overall performance of RMC across all versions, we switched to depth-bounded model checking for `Elevator` and `RaxExtended`.

For each version of evolution, the following steps were followed to obtain the data for our study:

1. We applied FMC to v_{new} and collected SS_{FMC}, ET_{FMC} and MU_{FMC}.
2. We applied recordingRMC to v_{old} to gather the data needed to perform the next step.
3. We applied pruningRMC to v_{new} and collected SS_{RMC}, ET_{RMC} and MU_{RMC}.

5.4. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object programs and versions. The programs are relatively small, and the versions are created through simulation. However, as discussed above, this simulation allowed us to attain greater internal and conclusion validity. Still, these threats need to be addressed by additional studies on different workloads.

The primary threat to the internal validity of this experiment is possible faults in the implementation of the algorithms, and in the tools that we use to perform evaluation. We controlled for this threat through the use of extensive functional tests on our tools and verification against cases in which we can manually determine correct results. A second threat involves inconsistent decisions and practices in the implementation of the algorithms studied; we controlled for this threat by having all of our algorithms implemented by the same developer (the first author), utilizing consistent implementation decisions and shared code.

Where threats to construct validity are concerned, the metrics we have chosen are important indicators of technique cost, but other metrics are also possible. For example,

Versions	Daisy			Elevator			AlarmClock			RaxExtended 10, 3, false			ReplicatedWorkers 5, 2, 0.0, 10.0, 0.05		
	State Ratio	Time Ratio	Memory Ratio	State Ratio	Time Ratio	Memory Ratio	State Ratio	Time Ratio	Memory Ratio	State Ratio	Time Ratio	Memory Ratio	State Ratio	Time Ratio	Memory Ratio
v1	1.04	0.90	1.48	1367187.00	50.50	2.45	1.00	1.00	1.25	1.86	2.09	0.36	2327.18	60.71	1.56
v2	1.00	1.02	1.82	1367187.00	51.19	1.27	1.15	1.33	1.24	4.26	4.57	2.15	646957.00	77.73	0.88
v3	1.01	0.94	1.35	2.02	2.16	0.62	1.00	1.33	1.26	2113379.00	137.82	1.16	1.00	0.89	0.81
v4	1.00	0.92	1.63	1367187.00	54.83	1.41	5.47	1.50	1.38	2113379.00	152.50	4.26	2432.18	72.83	1.80
v5	1.00	0.94	1.44	1367187.00	61.58	1.10	1.00	0.67	1.18	2113379.00	126.00	6.77	2432.23	65.46	1.73
v6	1.00	0.85	1.58	1367187.00	48.25	2.41	10.39	1.00	1.22	2113379.00	128.67	1.62	1.00	0.98	0.46
v7	48785.00	15.67	4.37	1367187.00	55.36	1.35	235.00	1.00	1.93	2113379.00	149.11	1.35	646957.00	84.20	1.45
v8	52.03	9.80	4.15	1367187.00	50.67	1.08	235.00	2.00	1.21	1.00	1.13	0.32	2405.07	78.64	0.82
v9	1.59	1.37	2.80	1984.20	47.47	0.90	10.39	1.00	1.22	19106.00	14.27	1.40	646957.00	78.20	1.60
v10	1.00	0.90	2.19	1367187.00	55.57	1.72	5.69	1.00	1.22	1.00	0.89	1.21	646957.00	74.91	1.50
v11	1.00	1.00	1.76	4.07	5.78	0.97	2.74	1.00	1.24	-	-	-	646957.00	76.73	0.92
v12	48785.00	15.33	7.00	1367187.00	52.70	1.65	8.24	1.00	1.29	-	-	-	1.15	1.22	0.70
v13	48785.00	15.00	7.26	1367187.00	44.61	1.45	1.92	1.50	1.23	-	-	-	1.03	1.04	0.58
v14	-	-	-	35.25	16.31	1.26	-	-	-	-	-	-	49766.23	74.09	0.87
v15	-	-	-	1367187.00	44.24	1.81	-	-	-	-	-	-	646957.00	79.67	1.53
v16	-	-	-	1367187.00	49.29	3.44	-	-	-	-	-	-	24883.12	77.09	1.54
v17	-	-	-	1367187.00	42.66	3.01	-	-	-	-	-	-	0.92	1.02	0.63
v18	-	-	-	1367187.00	40.84	2.75	-	-	-	-	-	-	2.19	2.79	0.50
v19	-	-	-	1367187.00	52.81	1.49	-	-	-	-	-	-	1.00	0.89	0.47
v20	-	-	-	1367187.00	47.50	2.49	-	-	-	-	-	-	1031.84	72.57	0.88
v21	-	-	-	1367187.00	51.61	1.17	-	-	-	-	-	-	0.85	0.90	0.83
v22	-	-	-	1367187.00	44.67	1.55	-	-	-	-	-	-	1.00	0.98	0.77
v23	-	-	-	1367187.00	45.10	1.54	-	-	-	-	-	-	646957.00	77.27	1.33
v24	-	-	-	1367187.00	46.03	1.57	-	-	-	-	-	-	646957.00	68.33	1.70
v25	-	-	-	1367187.00	43.44	1.40	-	-	-	-	-	-	646957.00	86.91	0.93
v26	-	-	-	1367187.00	53.17	1.50	-	-	-	-	-	-	0.75	0.76	0.49
v27	-	-	-	1.65	2.14	0.90	-	-	-	-	-	-	1.00	1.09	0.70
v28	-	-	-	1367187.00	46.59	1.75	-	-	-	-	-	-	1.00	0.93	0.26
v29	-	-	-	1367187.00	52.61	1.90	-	-	-	-	-	-	-	-	-
Average	11262.82	4.97	2.99	1131535.01	43.44	1.65	39.92	1.18	1.30	1058600.91	79.57	2.06	210996.56	43.53	1.01

Table 2. Summary of Results

our metrics do not factor in the human costs that would ultimately be associated with using the approaches. As a further matter, while our RMC technique preserves effectiveness with respect to FMC, and thus inherits the strengths of that technique for revealing various classes of faults in software, there have not yet been studies of such fault classes in the contexts of evolving systems. The ability for model checking techniques to reveal faults that arise in evolution contexts should be studied.

5.5. Results and Analysis

Table 2 summarizes our experimental results. The table presents data for each of the versions of our five object programs (93 total versions, ranging from 10 on `RaxExtended` to 29 on `Elevator`) along with overall (across all versions) averages.[2] In the table, for each version of each program, results are presented for each of our three dependent variables, but to facilitate analysis, rather than presenting raw data values, we use ratios comparing FMC and RMC. Thus, columns with header "State Ratio" represent the ratio SS_{FMC}/SS_{RMC}, columns with header "Time Ratio" represent the ratio ET_{FMC}/ET_{RMC}, and columns with header "Memory Ratio" represent the ratio MU_{FMC}/MU_{RMC}. Within these columns, values (ratios) greater than one indicate that RMC improved on FMC in terms of the corresponding cost, while values less than one indicate that FMC improved over RMC. Table entries of "-" indicate that there is no corresponding version for the corresponding object.

Where state ratios are concerned, we find a great deal of variation between different versions of a given object, which means that different modifications of the same program may benefit differently from the use of RMC in terms of new states visited. For example, on the 13 versions of `AlarmClock`, five have a state ratio less than two, four have state ratios in the range two to 10, and four have ratios greater than 10. We also find large variations between objects. For `Daisy`, nine of 13 versions have ratios less than two, which means that for most versions of `Daisy`, RMC provided relatively little improvement over FMC in terms of states visited. However, for `Elevator`, only one of 29 versions has a ratio less than 2.0, and 25 versions have ratios of 1984 or more. These indicate that the variation in savings

[2]The names of three object programs in the table are composed of the program name in the first line and (where applicable) a comma-separated list of parameter values for the program in the second line, indicating the configurations under which these programs were run.

of states by using RMC is enormous, but in general, RMC could provide substantial savings over FMC.

We note that in three of the versions of `ReplicatedWorkers`, v17, v21, and v26, the state ratio is less than one indicating that RMC visited more states than FMC. Analysis of the algorithm in Figure 4 reveals that this is impossible, but JPF implements a range of additional optimizations on top of DFS. Specifically, JPF's partial order reductions depend, in part, on the order in which states are visited which can vary between RMC and FMC. While this may effect many programs and versions, in only 3 of 83 cases does it result in a net increase in states visited.

As expected, the table also shows that time ratios often seem to increase in correspondence with state ratios. For instance, on `ReplicatedWorkers`, version v22 has a state ratio of 1.00 and a time ratio of 0.98, and version v5 has a state ratio of 2432.23 and a time ratio of 65.46. This relationship does not hold in all cases, however; for example, on version v24 `ReplicatedWorkers` has a state ratio of 646957.00 and a time ratio of 68.33, while version v20 has a (lesser) state ratio at 1031.84 and a greater time ratio at 72.57. While there is significant variation in execution time, the dominant trend is one of significant speedup; 59 of the 93 versions across all subjects show RMC halving the execution time of FMC.

It is interesting that different objects exhibit quite different memory ratios. On `ReplicatedWorkers`, memory ratios are small, which means that there is more memory cost associated with applying RMC than FMC. In contrast, on `Daisy` and `AlarmClock`, the memory ratio is larger. The ratio is around 1.0 for all versions of `AlarmClock`, which means RMC does not cost more in terms of memory than FMC in those cases. For all versions of `Daisy` the memory ratio is greater than 1.0, and on six of 13 versions it is greater than 2.0, meaning that RMC has lower memory cost in the critical period than FMC on these versions. As discussed in the next section, memory consumption in the preliminary period can be significant — in the case of `Daisy` the memory ratio drops from an average of 2.99 to 1.09 — however; RMC's pruning still results in an overall memory savings.

The comparison between time and memory ratios is also of interest; there appears to be no discernible relationship between them, and small memory ratios do not necessarily imply small time ratios. For example, version v9 of `Elevator` has a smaller memory ratio than version v10 of `Daisy` (0.90 vs 2.19), but a much bigger time ratio (47.47 vs 0.90). Moreover, many versions have small memory ratios while also having large time ratios.

Overall, we find that for almost all versions of the objects, RMC exhibits significant savings in numbers of new states visited and time cost. For some objects, RMC requires much more memory than FMC, while for some other objects, RMC requires the same amount of memory as, or even less memory than, FMC. However, for many cases in which RMC requires more memory, RMC still saves state and time cost as well.

Program	Time	Memory
Daisy	0.97	1.09
Elevator	1.57	0.36
AlarmClock	0.99	1.14
RaxExtended	1.56	0.75
ReplicatedWorkers	1.47	0.24

Table 3. Average Total Overhead

6. Discussion

Overall Performance. In our analysis we focused on only the critical period performance of RMC, but the overall performance of RMC could also be of concern in judging cost-effectiveness when preliminary period costs matter. To explore this issue we focus on memory usage and execution time, since the number of visited states for RMC is no larger than the number of visited states for FMC. To do this, we calculate overall overhead for RMC as the sum of the costs (in execution time and memory used) of running recordingRMC (RMC_r) on v_{old} and running pruningRMC (RMC_p) on v_{new}. For FMC, we calculate overhead as the sum of the costs of running FMC on both versions.

Table 3 provides a view of the results of these calculations, measured in terms of the ratios of costs of FMC to RMC. For execution time, the data shows that RMC exhibits a speedup on all five programs. This suggests that the cost savings in the pruning phase of RMC more than compensate for the increased cost incurred during the recording phase.

For memory usage, the overall performance of RMC is more variable. In two cases, `Daisy` and `AlarmClock`, the memory overhead for RMC is lower than that for FMC, while for the other three programs RMC uses more memory (in the worst case approximately 6 times more). While this overhead is significant one can view RMC as trading an increase in memory for a significant reduction in execution time in the critical period.

Optimizing Performance. We have not optimized our RMC prototype's performance; however, there are several potential ways in which to reduce its memory usage.

First, the structure of the program can be exploited to eliminate redundant reachable element information. For example, if reachable elements are basic blocks, dominator analysis can determine when the execution of one block implies that another block will be executed; we need only record the first such block. This space saving device will result in no reduction in pruning of states, but it may incur some time overhead since line 3a of DFS_p would involve checking the dominance relation.

Second, our algorithm admits a wide variety of coverage element types. In our study we explored only basic block elements, but coarser coverage elements, e.g., methods, offer the potential for significant savings. Unlike the dominance space reduction approach, when coarsening coverage elements there is a potential to negatively impact pruning. This is because a program execution may execute a method containing a change without executing a changed block, and in such a case a state may be falsely judged to be unsafe.

Limitations. Our current RMC approach tolerates arbitrary changes to the state space, but is sensitive to the structure of a program state. For example, if one were to refactor a pair of fields into a separate object but perform exactly the same computation, RMC would fail to match those states. The problem of pruning states when state structure changes is challenging, but one solution appears promising.

Explicit state model checkers such as JPF hash the state in an component-wise fashion, i.e., they hash parts of the state and store the relationship among those parts and then when a statement modifies the state they rehash only the affected portion. Currently, the decomposition of the state for hashing purposes is performed in a rather ad-hoc manner. It is possible, however, to use a type-based decomposition in which a single hash is computed for all instances of a Class. The advantage of this for RMC is that RMC$_r$ can record multiple type-specific hash codes for each state. Then, when a Class c is restructured, the hash codes for the remaining types can be used to match states during RMC$_p$ as long as there are no reachable elements in c.

Practical Implications. Our results have several implications for the practice of regression model checking and, more generally, validation of evolving systems.

First, our RMC technique relies on the RTS technique `Dejavu` to provide dangerous edge information. However, that technique itself helps engineers with regression test selection, and thus, its use integrates nicely into the RMC approach, and its cost can be amortized across the two tasks.

Second, we have restricted our attention to the process model presented in Section 2, but we believe our approach will also apply to other process models. For example, many software development organizations employ incremental maintain-and-test processes such as nightly-build-and-test, to help uncover system errors more frequently during the maintenance process. Here, both the preliminary and critical periods are shorter than in the batch model. It would be quite difficult in such situations to run full model checking on the entire system during the short critical period, but RMC may cut critical period costs sufficiently to allow its application, especially given that system deltas will be smaller in this case than under the batch process.

A third implication, and one related to the second, involves long-term use of RMC. The pruningRMC phase depends on the recordingRMC phase for data. As a system moves through a succession of releases, rather than applying a full version of RMC$_r$ to each release in order to enable the use of RMC$_p$ on a subsequent release, it would be preferable to incrementally compute only changed data, relying on data computed from historical releases where possible.

We believe that RMC can address these more general development contexts. The information recorded during RMC$_r$ can be accumulated over multiple version histories. To achieve this, one could define a variant of RMC that prunes the search based on recorded information and updates the reachable elements for states that are explicitly searched. When version deltas are small such a variant would perform significant pruning and minimal recording, and would likely exhibit very favorable time and space performance. Thus, the application of RMC may actually influence development practice towards more frequent and more localized versioning. In this way, the use of regression model checking may itself be an enabler for more frequent use of regression model checking, bringing its notable benefits into play across the entire lifetime of evolving systems.

7. Related Work

Much work has been done on validating evolving software using regression testing (e.g., [13, 17]. In the context of verification, there has been relatively little work.

Many researchers have explored frameworks for compositional or modular verification (e.g., [5, 8]). These approaches partition a system, either to reduce analysis costs or because not all of the system is available, and reason about each part using an abstraction of the behavior of the other part. In principle, this approach would be applicable to regression verification for systems built with behavioral interface contracts. Such contracts would serve to confine the impact of changes leading to reductions in analysis cost relative to FMC. Unfortunately, systems with contracts are not widely used – our approach is more general and more broadly applicable across software systems.

Hardin et al. [9] propose a type of regression verification that exploits the fact that specific correctness properties often relate only to a small portion of a systems behavior. For each property, the technique computes a localized abstraction and records a hash of that abstraction. When the system changes the abstraction hash can be recomputed and, if they hash is equal to the recorded hash the analysis of the property on the changed system can be skipped. The approach would work well if changes did not affect many properties, but there is no evidence that this is the case. Moreover, when a property does need to be re-analyzed there is no prospect for cost savings – our approach could be used in that case to reduce the cost of re-analysis.

In a short paper, Strichman et al. [20] propose the use of uninterpreted functions in reasoning about the functional

equivalence of program versions using bounded model checking. This approach will be effective only if the states on entry to the uninterpreted functions are equivalent, which is a strong condition that is unlikely to hold for realistic program changes. In contrast, our approach is not dependent on the relative equivalence of the program versions.

The work most related to this paper is incremental state-space exploration (ISSE) [12]. ISSE also focuses on incrementally evolving programs. ISSE stores the state space graph for a subsequent release of a system, and reduces the time necessary for state-space exploration by avoiding the execution of some transitions and related computations that are not necessary. To achieve this ISSE traverses the entire state space of the changed program while RMC skips safe sub-state spaces. We note that ISSE and RMC are orthogonal and the state processing reductions they achieve could be combined with the state-pruning reductions of RMC.

8. Conclusions and Future Work

We presented RMC, a technique for regression model checking, that applies model checking incrementally to new versions of systems. RMC reuses data obtained through model checking of earlier system versions, leveraging the facts that state spaces of consecutive versions tend to be similar, and that the influence of program changes tends to be localized within regions of a state space. We have implemented our RMC technique in JPF, and the results of an empirical study examining our RMC technique's effectiveness on a variety of programs and versioning changes show that RMC is significantly faster than traditional model checking, a result that should facilitate the use of model checking on programs as they evolve and throughout their lifetimes.

In the future, we plan to optimize the time and memory cost of RMC, and to investigate the effectiveness of RMC across different granularities of coverage elements and across sequences of versions of evolving systems.

Acknowledgments

The authors thank Peter Melhitz for his support of our work on extending JPF. This work was supported in part by the National Science Foundation under Awards CNS-0454203, CCF-0541263, and CNS-0720654 and by the National Aeronautics and Space Administration under grant number NNX08AV20A.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. Conf. Prog. Lang. Des. Impl.*, pages 203–213, June 2001.

[2] http://jakarta.apache.org/bcel.

[3] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proc. Int'l. Conf. Tools Algs. Constr. Anal. Sys.*, pages 73–89, 2006.

[4] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. Int'l. Symp. Found. Softw. Eng.*, pages 92–104, Nov. 2006.

[5] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. *Softw. Eng. Notes*, 26(5):152–163, 2001.

[6] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *J. Obj. Tech.*, 3(6), June 2004.

[7] A. Groce and W. Visser. Heuristics for model checking Java programs. *Int'l. J. Softw. Tools Tech. Trans.*, 6(4):260–276, Aug. 2004.

[8] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Prog. Lang. Sys.*, 16(3):843–871, Mar. 1994.

[9] R. H. Hardin, R. P. Kurshan, K. L. Mcmillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *Proc. WODES*, pages 147–150, 1996.

[10] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *Softw. Tools Tech. Trans.*, 6(4):302–319, 2004.

[11] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analyses for Java software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska - Lincoln, Apr. 2006.

[12] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. Int'l. Conf. Softw. Eng.*, pages 291–300, 2008.

[13] H. Leung and L. White. Insights into regression testing. In *Proc. Conf. Softw. Maint.*, pages 60–69, Oct. 1989.

[14] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. Symp. Net. Sys. Des. Impl*, Mar. 2004.

[15] http://home.att.net/d̄davies/NewSimulator.html.

[16] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. ACM*, 41(5):81–86, May 1988.

[17] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, Aug. 1996.

[18] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, Apr. 1997.

[19] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proc. Comp. Sec. App. Conf.*, pages 13–22, 2005.

[20] O. Strichman and B. Godlin. *Regression Verification - A Practical Way to Verify Programs*. Springer-Verlag, Berlin, Heidelberg, 2008.

[21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Auto. Softw. Eng. J.*, 10(2):203–232, April 2003.

[22] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. Symp. Op. Sys. Des. Impl.*, Dec. 2004.