Delta Execution for Software Reliability

Yuanyuan Zhou, Darko Marinov, William Sanders, Craig Zilles
Marcelo d'Amorim, Steven Lauterburg, Ryan M. Lefever and Joe Tucek
University of Illinois at Urbana-Champaign, Urbana, IL 61801

1 Introduction

1.1 Motivation

Software failures greatly reduce system reliability and availability, contributing to 26-30% of system failures according to several recent root-cause analysis studies [10]. Such software bugs can crash the system, making the service unavailable. Moreover, "silent" bugs that go undetected can corrupt information, generate wrong results, and lead to unavailability.

Besides software defects, administrative errors are another major cause for system failures. A recent study [14] has shown a significant fraction (37%) of failures in Internet services are caused by administrator mistakes. Among all factors, misconfiguration (mistakes made in setting configuration parameters) is a major root cause, contributing to 78% of administrative errors [12]. To quickly recover the system and repair the damage, service providers require many administrators to troubleshoot misconfigurations. As a result, system administration costs account for 60-80% of the Total Cost of Ownership in IT [12].

To improve software reliability and reduce administrative errors, many reliability assurance techniques have been used at different stages of the software life-cycle, including software testing, software patching, and online validation of administrative reconfigurations.

Interestingly, all these reliability assurance tasks exhibit a common characteristic: *multiple almost-redundant executions* (**MARE**). In other words, they all execute multiple versions/copies of the same software, each execution differing from the others only *slightly* in code segment, input, or configuration. Therefore, these executions are almost redundant with a few exceptions. We next explain in more detail the manifestation of MARE in each task.

MARE in software patch validation: To address software defects, especially those that can be exploited to launch security attacks, software companies frequently release patches to their software. Unfortunately, these patches are often buggy themselves because vendors are under pressure to release a patch quickly, without enough time for thorough testing [3]. For example, several recent Microsoft patches have caused major connectivity problems and resulted in substantial loss in business [13]. Due to this concern, administrators and users hesitate to apply patches, resulting in a longer window of vulnerability to software failures and even security exploits.

As many field experts warn, faulty patches can cause major financial damage to a business. The standard should always be to download, test, and then deploy patches as quickly as possible [4, 9]. Patch testing can be done either off-line [2] or on-line [6]. Even though off-line validation is simpler, on-line patch validation is much more accurate and desirable. This is because an on-line validation tests software patches against realistic and even live workloads, and thereby such validation can catch a larger proportion of mistakes.

During an on-line validation, two versions of the software are executed simultaneously. Both executions are fed with the same live inputs, but only the old stable version's outputs are visible. As software patches are usually small both in terms of modified code segments and the amount of touched data [11], the majority of code segments executed by the two runs are exactly the same. Therefore, most on-line validation of patches are MARE.

MARE in administrative reconfiguration validation: Since administrative reconfigurations can easily result in software failures or even permanent data corruption, a common practice is to validate the system after performing reconfigurations and before releasing them to production systems. Similar to software patch validation, administrative reconfiguration can also be validated either off-line or on-line, with similar advantages and disadvantages [12], i.e., on-line is more accurate because as it tests the new configuration against realistic and live workloads. Also similar to on-line software patch testing, on-line reconfiguration validation are also MARE, with each execution differing slightly in certain configuration parameters.

MARE in software testing: Software usually goes through several stages of testing before being released to customers. Testing is an important but also a very time-consuming part of validating software reliability. Each test case can execute a large portion of the program, which

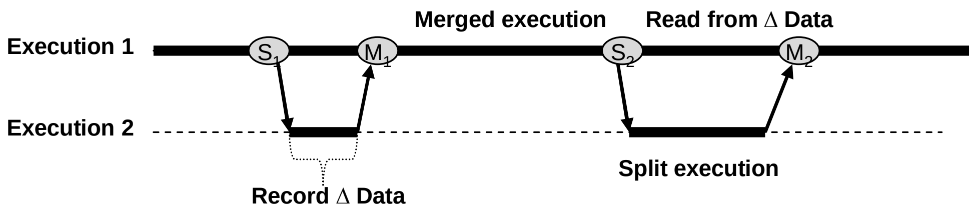


Figure 1. The delta execution idea. Two simultaneous, almost-redundant executions are separated only when they execute different code segments or access different data. Solid line segments indicate actual execution. S_1 and S_2 are split points, and M_1 and M_2 are merge points.

makes it expensive to test with many inputs required to achieve a good coverage. Also, it is necessary to test the program in different environments. Hence, many test executions are similar as they execute the same code for slightly different inputs or in slightly different environments. Thus, testing performs MARE. Another example of MARE arises in *regression testing*, when a new version of code is tested to behave the same as the old version, similar to the off-line validation of patches. Finally, software model checking is a form of thorough testing and can result in many similar executions (Section 5).

MARE in partial replication-based fault detection and recovery: Replication at the instruction, process, or system level has long been used to provide dependability. However, a major limitation to the adoption of replication has been the overhead incurred when doing replication, particularly if done at the process or system level. Another limitation of traditional replication is that it cannot detect common mode faults (e.g., software design faults, or bugs), since in that case, both replica would produce the same (wrong) result. To address this problem, recently we have developed a new replication approach known as *partial replication* [1]. In this approach, slightly different versions of code are executed that will produce the same result in the fault-free case, but will produce different results if a software design fault occurs. In this new promising recovery approach, multiple partially-different copies of the same software are executed simultaneously. In other words, it also performs MARE and would benefit from efficient support for performing MARE.

1.2 Current State of the Art

All of the above reliability assurance tasks involve MARE. An existing solution commonly used to fulfill this need is to fully execute these almost-redundant runs sequentially (as done for software testing) or simultaneously (as done for online patch and reconfiguration validation) on the same or different machines. The inherent inefficiency of this solution materializes in one of two ways: (1) *large overhead*: complete execution of MARE on the same machine would introduce large CPU and memory overhead to the production system, resulting in a significant decrease in throughput and increase in average response time. This is one of the primary reasons that prevents the prevalent usage of on-line patch and reconfiguration validations; (2) *large waste of resources and human effort*: executing MARE on separate machines requires doubling machine resources, as well as significant human administration effort to replicate the execution environments, resulting in a substantial increase in the costs of equipment, human operation, and energy consumption. Furthermore, both approaches negatively impact the validation time, by reduced throughput or increased machine set up time, respectively.

1.3 Our Idea: Delta Execution

To efficiently support the MARE used in performing various reliability assurance tasks, we propose a novel approach called *delta execution* and a comprehensive infrastructure to support it. The main idea of delta execution is to minimize the cost of MARE by sharing redundant execution and separately executing only *deltas*—code regions that run different code segments or process different data. Figure 1 shows two simultaneous runs that share most execution. The two runs split when they need to execute different code segments or read different data and then merge later when their executions reconverge to (at least locally) identical execution.

This position paper discusses the research challenges, design and implementation issues by using several real world patches as examples. Additionally, we have also conducted some feasibility studies of our delta execution idea on on-line patch validation and for improving model checking, and our preliminary analysis has shown some promising results.

Terminology In all multiple almost-redundant execution scenarios, the executions differ from each other in a small fraction of execution segments (called *delta segments*) that either execute private code segments (called *delta code*) or access private data (called *delta data*). Each delta segment starts with a *split point* and ends with a *merge point*. Initially, there is only one execution (called *merged execution*), i.e., the target program executes only one original copy of code, say the old stable version. At a split point, the single execution then splits into two or multiple, each executing its own private code and/or accessing its own private data. All these executions (called *split execution*) will later merge at a merge point, having their private data maintained separately by the underlying delta execution system.

2 System Overview

To support efficient delta execution, we need to address the following research challenges: (1) How to identify delta code? (2) How to identify split and merge points? (3) How to split and merge executions? (4) How to track and maintain delta data for each execution? (5) How to minimize time and space overhead? (6) How to apply and extend delta execution to support each specific reliability assurance task?

We address the above challenges by using a synergistic co-design of multiple layers including the operating and run-time system, dynamic and static compilation, programming language constructs, and application-specific extensions to support various reliability assurance tasks.

The merged execution splits because it needs to either execute delta code or access delta data. For the first case, the compiler marks the split points in the target program based on either static or dynamic analysis of the multiple executions. For the second case, the split point is triggered dynamically when the run-time system detects an access to private data. In both cases, the merge points are determined dynamically based on hints given by the compiler from static program analysis and dynamic profiling.

The run-time system provides several functions: (1) supporting splitting and merging of multiple executions; (2) supporting multiple concurrent executions by multiplexing inputs from other processes and external devices; (3) sandboxing side-effects made by selected executions; (4) maintaining delta data for each execution and detecting accesses to delta data during merged execution; (5) coordinating delta execution on multiple nodes for distributed applications.

Programming language support is needed for two purposes. First, it is needed to support application-specific run-time actions by allowing software to select which execution's side-effects should be sandboxed and to validate new execution against old. Second, it also provides simple language abstractions that allow programs to be written modularly to minimize the run-time overhead due to executing delta segments and maintaining delta data.

Extensions to support various reliability assurance tasks are specific to each task. For example, for software patch online validations, the compiler can relatively easily mark the delta code segments, i.e., those that are different between the old version and the patched version. But for online configuration validation, it requires more sophisticated run-time profiling to identify split and merge points.

Due to space limitation, the remainder of this paper will focus on the run-time system support and present some results on applying delta execution for improving software model checking.

3 Examples

This section uses three real-world software patches from the Apache Web Server (shown on Figure 2) as concrete examples to show how our proposed system would work and motivate the needs for advanced features. The first two examples represent the majority of software patches, whereas the last example gives a complex scenario that calls for advanced support for efficient delta execution.

Example 1: No state update. The first patch, which represents a large percentage of *security patches*, adds only a bound-check to prevent security attacks that exploit the buffer overflow bug. Therefore, in normal execution when the buffer overflow does not manifest, the patch only performs a bound check without any updates to the memory state. In rare cases when the buffer overflow is triggered, the patched version will deviate from the old version. As the patched version could be buggy (e.g., the wrong bound check or handling the buffer overflow incorrectly), it is important to validate it online before using it. Such a MARE task is a perfect candidate for our delta execution. Moreover, to further reduce overhead, our delta execution engine will use some light-weight system support to split, execute, and merge the two executions because, in this example, there is no delta data between the original version and the patched version, and the delta code is also very small during normal execution (no buffer overflow).

Example 2: With state updates. The second patch, which represents a large class of *bug fixes*, not only revises some code segments but also modifies a part of memory states. Therefore, validating this patch online using delta execution is more complex than the first type of patches, because the system needs to maintain and detect accesses to delta data. In this example, the compiler also needs to carefully select the split and merge points. If the compiler is conservative about merging, it can lead to fully redundant execution; if it is too aggressive, the execution will too frequently shift between split and merged execution, resulting in high overhead.

Example 3: With data structure layout change. The third patch, representing a small fraction of software patches, is the most complex of the three examples. It not only modifies the state, but also changes the data structure layout. The naive way to support online validation of this patch can result in almost fully redundant execution because the memory layout can become very different, resulting in large delta data and thereby large delta segments. A better alternative is to exploit the data encapsulation enabled by modern, type-safe languages to covertly allocate objects that maintain a union of the necessary fields. This example is more complex than patches that dynamically allocate

```
Patch: CAN-2004-0493
httpd-2.0/server/protocol.c, line #381
….

+       if ((fold_len - 1) >
+           r->server->limit_req_fieldsize) {
+               r->status = BAD_REQUEST;
+               return;
+       }
…
```

(a)

```
Patch: CAN-2004-0811
httpd-2.0/server/core.c, line #353
….
    for (i = 0; i < METHODS; ++i) {
        if (new->satisfy[i] !=
          SATISFY_NOSPEC) {
            conf->satisfy[i] = new->satisfy[i];
+       } else {
+           conf->satisfy[i] = base->satisfy[i];
        }
    }
…
```

(b)

```
Patch: for Bug #7067
experimental/cache_cache.c, line #80

Typedef struct{
…
cache_set_priority set_pri;
    cache_get_priority get_pri;
+   cache_cmp   cmp;
    cache_inc_frequency *inc_entry;
…
}
```

(c)

Figure 2. Real software patch examples from the Apache Web Server. Note that the code is slightly simplified for description purpose. Lines beginning with '+' are new code added by the patches.

some extra buffers. The latter case can be addressed by allocating the extra buffer from some isolated location to avoid artificially changing the heap alignment between the two executions.

Similarly, in other reliability-assurance tasks such as online configuration validation and software testing, different cases may have different levels of complexity in delta execution. While some cases only require simple methods to mark split and merge points (and light-weight system support for splitting, executing, and merging multiple executions), there are always other, more complex cases that require sophisticated program analysis and dynamic profiling and heavy-weight system support. Programming language constructs will be useful for such cases to improve the efficiency of delta execution.

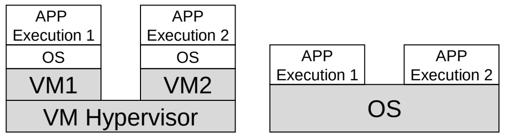
4 Design and Implementation Issues

Usage Models Our Delta Execution system supports two usage models shown in Figure 3. In the first model, each of the two executions runs on its own virtual machine, whereas the second model runs both executions on the same OS. Even though both models can sandbox the side effects of selected executions (e.g., the patched version), the first usage model provides a high level of isolation and is thereby more suitable for certain scenarios, such as the online validation of software patches, where security attacks are potential concerns. However, the first usage model is more heavy-weight and therefore may be overkill for many other scenarios such as software testing that do not necessitate such high level of isolation.

Merged execution. In the very beginning, multiple versions of the program are loaded. In the separate VM usage model, each execution starts on a separate VM. In the single OS model, each execution starts as separate processes. After being loaded, only the main execution (e.g., the old version) continues whereas the other executions (along with their VMs) are temporarily suspended until the next split point. At any moment during the merged execution, the global state and the memory updates are shared among all executions. Shared virtual pages in multiple executions are mapped to the same physical memory so that updates to these shared data are visible to all executions. Such mapping would require support from the VM hypervisor similar to Introvirt [9].

Split execution. At a split point, each suspended execution continues by first copying the execution environment such as stacks from the merged execution and then starts executing from the split point. During the split execution, each execution runs on its private stack and either executes its private code or accesses its private data. Any updates made during split execution are maintained as delta data. Any deallocation of a delta data (including stack frame pop-out upon function returns) will remove the content of such data from the private data pool. The split execution continues until the next merge point, and then all executions except one are temporarily suspended again. Any shared data updated during split execution are removed from the shared pool into the private pool and are no longer shared by multiple executions.

Memory layout. All executions share the same memory manager for allocation and deallocation, so it is possible to match the memory layout in each execution as much as possible. During the merged execution, the memory manager allocates buffers from the shared free memory pool. During the split execution, by default, the memory manager allocates from the private free memory pool associated with



(a) Separate VM Model (b) Single OS Model

Figure 3. Two usage models supported by our proposed Delta Execution system.

each execution. To reduce the amount of delta data, we will also explore optimization techniques to align buffers allocated during split execution.

Detecting accesses to delta data. The merged execution needs to split at an access to delta data. Therefore, we need to detect accesses to delta data during merged execution. One method is to instrument each memory access to check if this access is to delta data or shared data. This method would incur large overhead. Another method is to use page protection to monitor accesses to delta data. But due to the large page granularity, the method can result in significant false sharing. To avoid false sharing, we can allocate each delta object in a separate page or use ECC memory as in our previous work [15] to monitor accesses to delta data at cache-line granularity.

Light-weight support for small delta segments For simple cases such as Figure 2(a), using the full Delta Execution system support described above can be unnecessarily heavy-weight. Therefore, to handle simple cases, we can run multiple executions one after another in only *one* stack. After executing one version to the merge point, it records the updates made during the execution in this version's private data area and then rolls back the updates to run the next execution, and so on until all executions reach the merge point. For cases that do not have any memory updates as in example shown in Figure 2(a), this approach would work extremely well because it can completely avoid the overhead of stack copying described above. But for complex cases, this optimization is not good because it serializes all executions and thereby cannot take advantages of multiprocessor systems.

Optimizations to reduce split and merge overhead. Split and merge operations incur overhead, so performing splits and merges frequently can hurt performance. In some cases, the merge point can be misplaced so that immediately after such a merge point, the execution needs to split again. To optimize, the system can remember this mistake and inform the dynamic program analyzer to postpone a merge point until some subsequent program locations. Moreover, the system can even do some cost-benefit analysis to estimate whether it is benefitial to merge or just rollback to the end of previous split execution.

Handling file I/Os. File updates are also handled in a similar way to memory updates using shared file blocks and private file blocks. An access to a private file block also triggers execution split. Any file blocks updated during split execution also become private file blocks. Such functionality can be achieved by leveraging versioning file systems. For on-line patch and reconfiguration validations, only those updates made by the old version are committed to permanent storage.

Handling other I/Os and asynchronous events. External inputs from keyboard, networks, and other processes are fed in the same order to all executions during split execution periods. To reduce the amount of delta data, we propose to delay the delivery of asynchronous events until the next merge point or the threshold of delay time expires. For external outputs to monitors, networks, and other processes, the application can select which execution's outputs can be made visible to the outside world.

Extension for Distributed Applications For distributed applications where multiple nodes require MARE, messages sent during split execution from one node, say node 1, to another node, say node 2, should trigger execution split on node 2. This is because different versions on node 1 may generate different message content. As a consumer of this message, node 2 should also split the execution with each execution receiving the message from the corresponding execution on node 1. Other than this, the execution on each node can split or merge independently, even though coordinating the merge point can reduce the amount of delta segments thus improving delta execution efficiency. Ideas can be borrowed from coordinate checkpoints [5, 8] to explore such optimization options.

5 Preliminary Results

We have conducted two studies to evaluate the feasibility of our delta execution idea on software testing [7] and software patch validation.

5.1 Software testing (Model Checking)

We have evaluated the use of delta execution in explicit-state model checking. Specifically, we use a model checker to perform bounded-exhaustive testing of object-oriented programs. In this testing scenario, a subject class is exercised with sequences of method calls up to a given length. A complete exploration considers all possible method orderings, as well as all variations of method arguments selected from a bounded set of input values. Despite a model checker's ability to avoid repeated exploration of previously seen states, this form of testing involves a *vast number* of MARE and is an ideal target for delta execution.

We have compared the performance of a model checker operating mormally and one using delta execution. Experiments were conducted using Java PathFinder (JPF) [17], a model checker for Java bytecodes. The exploration described above is typically performed in a breadth-first fashion, one method call at a time. Thus, we modified the model checker to execute a method against sets of program states simultaneously, splitting execution when necessary, and merging the states that are generated at the end of each level of the search: the exploration of level $i + 1$ is performed with the new states produced in level i.

We first evaluate the delta execution model checker using 10 subjects taken from a variety of sources and used previously in other studies of testing. Table 1 shows for

experiment		JPF time			JPF mem.
subject	N	std	delta	std/delta	std/delta
binheap	7	25.40	2.66	9.55x	1.16x
	8	466.00	15.34	30.37x	1.03x
bst	9	44.34	10.98	4.04x	0.70x
	10	216.72	49.17	4.41x	0.46x
deque	8	54.86	6.64	8.27x	1.50x
	9	550.57	57.72	9.54x	1.48x
fibheap	7	24.88	3.13	7.94x	2.13x
	8	398.13	28.31	14.06x	0.88x
filesystem	3	2.03	1.98	1.03x	0.97x
	4	17.13	3.70	4.63x	11.50x
heaparray	8	104.50	4.18	24.99x	2.31x
	9	2,718.12	26.96	100.81x	1.22x
queue	6	7.76	1.62	4.79x	2.64x
	7	104.41	6.37	16.38x	1.77x
stack	6	4.95	1.46	3.38x	1.01x
	7	59.44	5.08	11.71x	1.31x
treemap	10	579.50	7.61	76.14x	2.69x
	11	1,754.34	19.42	90.34x	3.04x
ubstack	8	60.37	6.26	9.64x	1.57x
	9	1,482.75	48.75	30.41x	1.48x
gmean	-	-	-	**10.97x**	**1.51x**

Table 1. Experiments using delta execution for model checking.

each of these subjects, the bound in the sequence length (i.e., the number of method invocations issued against the subject under test), the time it takes to run the experiment using the standard JPF tool, the time it takes to run the experiment using the modified JPF tool for delta execution, and the ratio of time and memory. Our initial findings show that on average delta execution can speed up the exploration time an order of magnitude while consuming 1.5 times less memory than the standard exploration.

We also evaluate delta execution by model checking an implementation of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol, a larger case study involving 43 Java classes and over 3,500 non-comment, non-blank lines of code. This implementation was previously used to evaluate other model checkers [16]. The results show that delta executions speeds up the exploration of the AODV state space to 1.43 times.

5.2 Software Patch Validation

We have conducted a preliminary investigation using two different Apache Web Server patches to evaluate the feasibility of our delta execution idea. We instrument the binaries of the old version and the patched version using an instrumentation tool called Pin from Intel. To compare the difference in code executed, we collect the execution trace from each run and compared their differences. The differences in data are computed using the checkpoint-rollback support in Pin and we report the largest differences during the entire execution. Each experiment is repeated many times and the result is stable with very small error rate caused by a small degrees of non-determinism. We have also conducted similar experiments for several other Apache and MySQL patches, and the results fall between the two extremes shown here.

Versions	2.0.44	2.0.50
Dynamic Delta Code	0.6%	2%
Dynamic Delta Data	2%	12%

Table 2. Preliminary feasibility results.

Table 2 shows the amount of delta code and data of each patch compared to the corresponding old version. The results show that the patched execution differs from the old execution only slightly, smaller than 2% for code and 12% for data, which indicates that our delta execution idea is feasible and should significantly improve the resource and time efficiency of MARE performed by various reliability assurance tasks.

References

[1] V. S. Adve, A. Agbaria, M. A. Hiltunen, R. K. Iyer, K. R. Joshi, Z. Kalbarczyk, R. M. Lefever, R. Plante, W. H. Sanders, and R. D. Schlichting. A compiler-enabled model- and measurement-driven adaptation environment for dependability and performance. In *Proceedings of the Next Generation Software (NGS) Workshop at the International Parallel & Distributed Processing Symposium (IPDPS'05)*, April 2005.

[2] R. Barrett, P. P. Maglio, E. Kandogan, and J. Bailey. Usable autonomic computing systems: The administrator's perspective. In *ICAC'04*, 2004.

[3] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the application of security patches for optimal uptime. In *LISA*, pages 233–242, 2002.

[4] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen. A trend analysis of exploitations. In *IEEE Symposium on Security and Privacy*, 2001.

[5] Y. Chen, J. S. Plank, and K. Li. Clip: A checkpointing tool for message-passing parallel programs. In *SC*, 1997.

[6] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *ICSE*, pages 203–212, 1999.

[7] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA*, July 2007. (To appear).

[8] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing system. Technical report, TR CMU-CS-96-181, Carnegie Mellon University, 1996.

[9] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *SIGOPS Oper. Syst. Rev.*, 39(5):91–104, 2005.

[10] E. Marcus and H. Stern. *Blueprints for High Availability*. John Wiley & Sons, 2000.

[11] Microsoft White Paper. Microsoft patch management summary, 2003. http://download.microsoft.com/ documents/australia/ business/mes/agenda/patch_mgmt-final.doc.

[12] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, 2004.

[13] R. Naraine. Faulty microsoft update rekindles patch quality concerns, 2005. http://www.eweek.com/ article2/0,1895,1815956,00.asp.

[14] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it, 2003.

[15] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *HPCA'05*, Feb 2005.

[16] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *ICFEM*, 2005.

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.