Extending Model Checking with Dynamic Analysis

Alex Groce and Rajeev Joshi

Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA *

Abstract. In *model-driven verification* a model checker executes a program by embedding it within a test harness, thus admitting program verification without the need to translate the program, which runs as native code. Model checking techniques in which code is actually executed have recently gained popularity due to their ability to handle the full semantics of actual implementation languages and to support verification of rich properties. In this paper, we show that combination with dynamic analysis can, with relatively low overhead, considerably extend the capabilities of this style of model checking. In particular, we show how to use the CIL framework to instrument code in order to allow the SPIN model checker, when verifying C programs, to check additional properties, simulate system resets, and use local coverage information to guide the model checking search. An additional benefit of our approach is that instrumentations developed for model checking may be used without modification in testing or monitoring code. We are motivated by experience in applying model-driven verification to JPL-developed flight software modules, from which we take our example applications. We believe this is the first investigation in which an independent instrumentation for dynamic analysis has been integrated with model checking.

1 Introduction

Dynamic analysis [1] is *analysis of a running program*, usually performed by the addition of instrumentation code or execution in a virtual environment [24]. Model checking [4] is a technique for exploring all states of a program's execution space, which may be a static analysis of an extracted model, as in CBMC [17] or SLAM [2], or a dynamic analysis in which a program is executed, as in CMC [22] or SPIN's *model-driven verification*. In model-driven verification [16] (our focus in this work) a harness embeds code and the model checker runs the program being verified in order to take a transition in its state space.

* The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was also provided by NASA ESAS 6G.

In this paper we show that the power and ease of use of model-driven verification can be significantly enhanced using dynamic analysis. Our approach extends the capabilities of the model checker by introducing instrumentation into the executed code — instrumentation which interacts with the model checker to perform property checks, modify control flow of the program, compute coverage metrics, and to guide the state-space search. Our approach is motivated by our experience with model-driven verification. In particular, our frustration when debugging model-driven verification harnesses and our interest in properties requiring fine-grained observation or control of execution suggested the use of automated instrumentation. Our examples are taken from JPL flight software produced or tested by our group: **NVDS**, a module used to store critical spacecraft parameters on flash [10] (5K lines of C for the module, plus 2K lines for a harness/reference), **Launchseq**, a model of the launch sequence for a recent mission (1K lines), and a replacement for `string.h`. Another motivation was our interest in random testing [10]: using independent dynamic analysis (rather than modifying SPIN), we are able to use the *same instrumentation* during a model checking run and a random test execution. We simply instrument and compile the program to be tested, and link to the same instrumented binary.

A further interest in such a combination is based on a common objection to dynamic analysis: it is fundamentally unsound. Model checking offers the possibility of combining a dynamic analysis with a complete exploration of the reachable state space (or an abstraction of that state space). It is true that, for realistic programs, the state space is often too large for an exhaustive search (even after abstraction); it is also often the case that analysis for a particular property adds variables to the program state and increases the state space size. Nonetheless, in some non-trivial cases, model checking offers an easy path to sound dynamic analysis.

Below, we discuss the particular instrumentations we developed. We first present a method for checking modifies clauses, also useful in debugging model-driven verification harnesses (Sections 4.1 and 4.2). We then show how the same framework supports a method for simulating software resets (Section 4.3), better coverage measures during model checking (Section 4.4) and, perhaps most interestingly, a novel search approach based on path coverage information produced by instrumentation (Section 4.5). Experimental results confirm our intuition that overhead will be low for most of these approaches (Section 4.6).

2 Model-Driven Verification

Model-driven verification [16] is a form of software model checking that works by executing code embedded in a model[1]. The SPIN model checker [14] operates by translating a model written in PROMELA into a (C) program to model check that program: in a sense, SPIN is less a model checker than it is a *model*

[1] In the current implementation, the executed code must be C, but this is not a fundamental limitation of the technique; in fact, we occasionally make calls to C++ functions in embedded C code.

```
1   c_decl {
2     extern struct s_t *arr;
3     extern int cnt;
4     extern int new_n(void);
5     int tmp;
6   };
7   c_track "arr"     "sizeof(struct s_t) * 100"    "Matched"
8   c_track "&cnt"    "sizeof(int)"                 "Matched"
9   c_track "&tmp"    "sizeof(int)"                 "UnMatched"
10
11  int r;
12  active proctype harness () {
13    c_code { cnt = 0; };
14    do
15    :: c_expr {cnt < MAX} -> c_code { tmp = cnt; now.r = new_n(); };
16                             assert(r != -1);
17                             assert(c_expr {cnt == tmp+1})
18    :: else -> break
19    od
20  }
```

Fig. 1: PROMELA Model with Embedded C Code

checker generator. Model-driven verification exploits this fact to embed C code within PROMELA models[2]. With model-driven verification, it is possible to check a C program against specifications written in linear temporal logic, using all the features of SPIN, including bitstate hashing, abstraction, and multi-core execution [14, 16, 15].

Figure 1 illustrates the use of SPIN's primitives for embedding C code. The `c_decl` primitive is used to declare external C types and data objects that are used in the embedded C code. The `c_track` declarations are *tracking* statements, which provide knowledge to SPIN of what state is being manipulated by the C program. We describe `c_track` statements in more detail below. The PROMELA process defined by the `proctype` declaration also uses the `c_expr` construct to embed C expressions that are used as guards and the `c_code` construct to embed blocks of C code within the PROMELA model.

During its depth first search[3], the model checker may reach states with no successors (e.g., the `break` from the `do` loop on line 18, which leads to a final state) or states that have already been visited. In such cases, the model checker backtracks to an earlier state to explore alternative successors. For variables in the SPIN model, such as `r`, restoration of an earlier value when backtracking is automatic. In order to restore data objects of the C program, however, the model checker needs knowledge of the set of memory locations that can be modified by the C code, which we call the *tracked* locations. For each data object modified

[2] This feature was introduced in SPIN version 4.0

[3] Note that SPIN currently supports embedded C code only with DFS.

by the C program, a `c_track` statement is used (see lines 7-9) to indicate three pieces of information needed by SPIN: the starting address in memory where the data object is stored, the size (in bytes) of the C representation of the data, and whether or not the data should be *matched*, i.e., whether it should be included in the check determining if a state has been seen before.

2.1 Tracking and Matching

It is important to note the distinction between tracking and matching. Tracked data objects are stored as part of the state on the stack used by depth first search (DFS). This allows SPIN to properly restore data values on each backtracking step during the DFS. As a rule, *all* data objects that can be modified by a C program should be tracked (with some exceptions, as discussed below).

Matching, on the other hand, allows SPIN to recognize when a state has been seen before. The set of matched data objects therefore constitutes the state descriptor, which is examined whenever a state is generated, to determine if the state has been seen before. The `''Matched''` and `''UnMatched''` keywords are used in a `c_track` declaration to indicate whether an object is matched or not.

Since the amount of data modified by a C program can be large, declaring all data objects as matched makes the state descriptor very large, and increases the size of the state space to be explored. In such cases, careful distinction between matched and unmatched data allows on-the-fly abstractions to be applied during model checking. A simple example is symmetry reduction: e.g., if program states are equivalent (with respect to verification of a property ϕ) up to any ordering of the items in a linked list, we may track the concrete variable but match only on a sorted version of the list, greatly reducing the size of the state space needed to verify ϕ. This approach to abstraction is discussed at length in the original paper on model-driven verification [16].

Note that not all data needs to be tracked. Data that does not change after a deterministic initialization process, or data that is not relevant to the search, does not require tracking. We refer to such data as *ignored* data.[4] There is no memory overhead for ignored data, but of course such data is not restored when backtracking occurs. Program state that is not modified in a way that is visible to SPIN can be ignored. It is also important to ignore memory that stores cumulative or statistical information over an entire model checking run.

2.2 Limitations of Previous Work

Each fragment of C code embedded in a SPIN model (using either the `c_expr` or `c_decl` constructions) is executed as a (deterministic) atomic step. This leads to several limitations of the SPIN approach to model-driven verification: (a) we cannot check properties (such as program invariants) within embedded C code, (b) we cannot interrupt control flow within a C function (for instance to simulate

[4] There are also situations where it is useful to declare *matched* data that is *untracked*; however, these are beyond the scope of this paper.

an asynchronous interrupt or an unexpected reset), and (c) we cannot interleave different fragments of C code (to check multithreaded C programs).

In this paper, we discuss how to address the first two limitations by using *program instrumentation*. In particular, we describe (i) how we check properties within C code, for instance on every write to global data, (ii) how we check C programs against unexpected events, for instance a *warm reboot* in which the program stack is cleared, but global data and the heap are not affected, (iii) how we can dynamically check *modifies clauses* [19], which constrain what data can be modified by a C function, (iv) how we can compute various *coverage metrics* (such as predicate coverage) of a C program over a model checking run, and (v) how we can dynamically apply various (sound and unsound) abstractions (for instance, a dynamic form of path coverage).

3 Dynamic Analysis via CIL Instrumentation

We insert instrumentation for dynamic analysis via source-to-source transformation. Our applications do not involve binaries without source, and we enjoy the benefits of adding instrumentation before optimization. Running the model checker itself under instrumentation is too expensive (and in some cases impossible), and it is very easy to instrument only certain compilation units. Our interest is in the use of instrumentation for analysis during model checking, not in the specific method used for inserting instrumentation.

3.1 Instrumentation with CIL

CIL (the C Intermediate Language) is a high-level intermediate language for representing C programs, and includes a set of tools that enable analysis and transformation of C programs [23]. CIL rewrites C programs in a semantically clean subset of C. User-written modules may modify the code as it is rewritten. The CIL distribution includes modules providing points-to analysis, array heapification, and other useful transformers and analyses. We use CIL because we find it to be a robust and easy-to-use tool for C source-to-source transformations.

Most of our analysis tools are adapted from the `logwrites.ml` module provided with CIL. This module "logs" all writes to non-stack-local memory, seen in CIL as `Set` or `Call` instructions. Because CIL analyzes program source, it can conservatively avoid instrumenting writes to stack local variables in a function.

```
void checkWrite (void *p,        /* Address of the memory */
                 size_t size,    /* Size (in bits) of the write */
                 const char* lv, /* Pretty-print of source lval */
                 const char* f,  /* Name of the file */
                 unsigned int ln /* Line number of the write */ );
```

Fig. 2: Prototype for `checkWrite`

```
in_stack(p) = ((p > stack_beg_loc) && (p < &stack_end)) ||
              ((p > &stack_end) && (p < stack_beg_loc))
```

Fig. 3: Definition for `in_stack`

Our adaptation is to change `logwrites.ml` to call, in place of a logging function expecting a string, a function `checkWrite` that expects more information. The prototype for `checkWrite` is shown in Figure 2.

3.2 Tracking the Location of the Stack

Our most common instrumentation involves checking writes to global memory. CIL distinguishes between local and global variables when this is possible, but cannot statically determine if certain pointers always target the stack. In order to determine the location of the stack, we add a global variable (`stack_beg_loc`) to the model checker, containing the address of a local variable of the `main` function, and declare another local variable (`stack_end`) in the scope of `checkWrite`. We assume that stack variables lie in the region formed by these boundaries, and define `in_stack (p)` to handle different stack orientations (Figure 3).

3.3 Replacing Memory Modification and Allocation Library Calls

Unfortunately, accesses visible to CIL as `Set`s and `Call`s to lvalues do not capture all memory writes. C programs also modify state by calls to system libraries — in particular, by using `memset`, `memcpy`, `memmove`, and the destructive string library functions (`strncpy`, `strcat`, etc.). We do not wish to recompile these libraries with CIL, but do wish to instrument the writes they produce. We therefore use another CIL module to rewrite these calls, making the memory writes visible.

We use a similar CIL module to replace calls to the `malloc` family with calls to `spin_malloc`, in order to make dynamic allocation visible. The `spin_malloc` functions use a static region that is tracked. This method also optionally provides checks for common memory-safety properties (no use after `free`, etc.) and ensures that tracked and allocated regions are equivalent if they overlap.

4 Applications and Experimental Results

We now present the uses we have made of dynamic analysis during model checking, and present experimental results indicating the utility and efficiency of our approach. Significantly, we show that the *relative* overhead for our instrumentation is quite low: the model checking engine is not instrumented, and tends to consume a large portion of runtime during model-driven verification.

Our applications include novel ideas specific to model checking (Sections 4.2 and 4.5). We also present more common analyses applicable in testing, in order to show the degree of reusability provided by independent analysis and to

compare analysis overheads for testing and model checking. The range of possible applications is potentially that of most runtime analyses for C programs.

4.1 Checking Modifies Clauses

Modifies clauses are used in ESC/Java [8], JML [3], Larch [27], and other languages to specify which variables a function may alter. We take a lower level approach and consider a specification of which *memory locations* a C function or block may change. These (named) locations are specified as a set of ranges, which may be dynamically computed during execution. A `checkWrite` function determines the correctness of each memory write:

```
void checkWrite (void* p, ...)
  forall (range ∈ modifiable_ranges)
    if (p ∈ range ∧ allowed(range))
      return;    /* Ok, p is in a modifiable range */
  /* p is not in any modifiable range! */
  ERROR;
```

A range is not statically defined, but is a dynamically evaluated region specified by expressions for starting and ending address. Whether a range can be written to can by dynamically toggled, depending on conditions. E.g., the stack will typically be included in the allowed ranges, but not in all cases. In our test harness for a replacement version of the C string library used in a JPL flight module, the `r_strncat` range is computed based on the arguments to the `n_strncat` function — `r_strncat = (t+strlen(t), t+strlen(t)+n)`. Code that calls `n_strncat` (the module's version of `strncat`) is rewritten with CIL to set up the restrictions:

```
disallow_all_ranges(); /* Clear set of modifiable ranges */
allow_range(r_strncat); /* Allow range for n_strncat */
char *result = n_strncat(t, s, n);
```

An advantage of our approach to combining model checking and dynamic analysis is that the analysis can be used in other kinds of testing or as a runtime monitor in deployment. We used the same instrumentation to check modifies clauses in a randomized differential testing harness [10] for the string library, comparing results of operations on randomly generated strings to those returned by the standard string library. These tests detected a minor error in argument checking in one function.

Extending Modifies Clause Checking to Library Calls In addition to restrictions on memory writes, we also support limitations on which libraries can be called. The most common application may be to restrict write access to devices (such as flash storage in our case) accessed through driver calls. In addition to device access properties, this also allows us to check performance properties, e.g. that no expensive library calls occur while interrupts are disabled on the flight CPU.

4.2 Debugging SPIN Models with Embedded C Code

One application of modifies clause checking is to assist in developing the harness for model-driven verification. A common error in such cases is to leave an important variable untracked, resulting in spurious counterexamples when the model checker backtracks but only partially restores a previous state. Our approach to debugging memory tracking during model-driven verification is to automatically generate a `checkWrite` function from the `c_track` statements in the SPIN harness. Our tool also supports a `c_ignore` statement, used to indicate modifiable memory that does not require tracking. During model checking, `checkWrite` acts as a modifies clause checker, ensuring that the program being verified does not modify any locations that are not tracked or ignored.

If the model checked program does write to untracked/ignored locations, SPIN will produce a warning for each such write — e.g. in the NVDS example:

```
UNTRACKED WRITE: 0x73b980 (nvds_npart) at nvds_init.c:377
UNTRACKED WRITE: 0x73b9d0 (nvds_ptsem[pt]) at nvds_init.c:258
```

In addition to `c_track` and `c_ignore`, we support a `c_constant` declaration. Unlike the C `const` type attribute, this does not indicate that an address is never assigned to (in which case we could simply leave the value out of our declarations). Instead, it produces a check that the value written to an address is always the same as the previous contents of that address. Because many of our models include simulations of system resets, we often call initialization code (such as would be called when a spacecraft reboots) setting global parameters, such as the size of a file system's memory region. The `c_constant` declaration provides warnings if this initialization code is faulty and changes the previous value of such a parameter, while avoiding spurious warnings about non-modifying assignments. Such declarations incur the additional overhead of a `memcpy`, but only before assignments to these addresses.

We used this approach to detect three untracked writes in a SPIN harness for a critical piece of flight software (the NVDS system), and to confirm that these writes were safe. We also verified the tracking of state for launch sequence modeling code derived from an upcoming mission.

4.3 Simulating Warm Resets

Another application of instrumentation is to return control to the model checker to simulate system resets. This is useful in two cases: simulation of cold resets on systems with a persistent hardware state that survives reboot, and simulation of *warm resets*, used in some experimental flight software at JPL.

In a warm reset, all data on the program stack is cleared and the program is restarted, but global data is *not cleared*. In some applications, this memory may need to be recovered, if possible, even after the software has been terminated in mid-operation and re-started. Because stack memory is lost on a reset we can reduce the state space of possible reset points by only considering resets at global writes. Again, we make use of a `checkWrite` function:

```
if (in_stack(p))
  return;    /* p on stack, no need to consider reset */
if (reset_trap > 0)
  reset_trap--;
else if (reset_trap == 0)  /* Trap goes off, reset */
  reset_trap = -1; /* Clear the trap */
  longjmp (context, code);
```

As with modifies clauses, this `checkWrite` requires the model checker to set certain variables before calling the tested code. In particular, it expects `reset_trap` to indicate if a warm reset is scheduled (-1 indicates no trap, a positive number n indicates that a warm reset is to take place on the nth write). A `setjmp` call to establish the `context` for `longjmp` to control to SPIN is also required. The model checking harness places resets nondeterministically.

We use this module in both model checking and random testing. The method has exposed subtle errors, including a very low probability scenario arising from the precise placement of a reset during a `memcpy` in a spacecraft RAM file system — a checksum used to detect memory corruption was too weak [10]. Detecting the error required a precisely placed reset during a memory copy.

4.4 Granularity of Coverage Measurements

Dynamic analysis also allows us to compute (abstraction) coverage at a finer granularity than atomic step boundaries. In our model checking of flash storage systems we use a number of unsound abstractions, as the state spaces for the rich properties we wish to check (essentially full functional correctness) are not amenable to sound abstraction and are large for even small flash devices. We may abstract the state of the flash device by only considering, e.g., the *state* of each block on the device (used, free, bad) and the number of live, dirty, and free pages on that block [25, 10]. When SPIN reaches a state in which the abstract state has previously been visited, it will backtrack. Because it may not be possible, under this abstraction, to reach all abstract states (indeed, certain states are defined as errors), we compute the coverage in cumulative fashion as we model check the file system. Unfortunately, computing coverage after every call to the file system does not measure actual abstraction coverage. Before an operation returns control to the harness, it may perform multiple writes to the flash device. In this case, our coverage is a measure of “stable states” of the flash with respect to the storage API, but is an underestimate of all covered states. We remedy this by instrumenting all driver calls for the flash device to recompute coverage after each modification. Again, this coverage instrumentation is as useful in randomized testing as in model checking.

4.5 Using Local Path Coverage to Guide Exploration

Another useful CIL module instruments every branch choice to update a bit vector representing the program path as shown in Figure 4. Before each entry

```
if (buff == NULL) {
   return;
} else {
   copy(x, buff);
}
```

(a) Before

```
if (buff == NULL) {
  add_to_bv(pathBV, 1);
  return;
} else {
  add_to_bv(pathBV, 0);
  copy(x, buff);
}
```

(b) After

Fig. 4: Before and after insertion of path tracking.

into the tested program the model checking harness clears `pathBV`. At the end of each call, `pathBV` contains information sufficient to reproduce the control path taken by the tested function, e.g. [] for branch-free code, and [1] or [0, ...] for our example, (where ... represents any branching taken inside `copy`). We limit the size of `pathBV` in some fashion — in our case, simply by taking only the first k bits of history, though other schemes, such as a sliding window, might also yield useful results.

Making `pathBV` a matched location adds path coverage of tested functions to the state space abstraction used in SPIN. This provides no benefit if we are matching on *all* aspects of the program state, but produces a new exploration strategy when combined with coarse abstractions. Consider the extreme case where we match on *no* program state. Without path information, no path will involve more than one entry-point function from the test-harness loop and will never discover any error requiring more than one function call. However, if we match on `pathBV`, SPIN will explore deeper paths, until such exploration fails to cover new paths in the tested functions. In the extreme case, this is unlikely to provide significantly deeper coverage, but for even very coarse abstractions may reach deeper state without the need to guess what additional program state should be tracked.

We applied this approach to our NVDS example, removing all matched state (other than the SPIN variables controlling inputs to tested functions) from the model, and adding matched path information. We ranged k from 0 to 20 bits (at 20 bits, the state space was large enough to exhaust memory on our 8GB machine). As expected, statement coverage of the module increased monotonically with the number of bits – but only by a few statements. The number of states explored increased dramatically — from 607K states for the 0-bit (no coverage) experiment to 48,100K states with 20 bits of path information. Most interestingly, coverage of the abstraction discussed above, approximating the physical state of the flash device (with respect to storage semantics of live and dirty pages) also increased. This increase was not monotonic, but showed a general upwards trend with k, ranging from 39 states at 0 bits to 53 states at 18 bits (falling back to 52 states at 21 bits). Matching the abstract state itself (and thus preventing backtracking whenever a new abstract state was reached) only improved on this by one state, covering 54 of the 55 reachable abstract states.

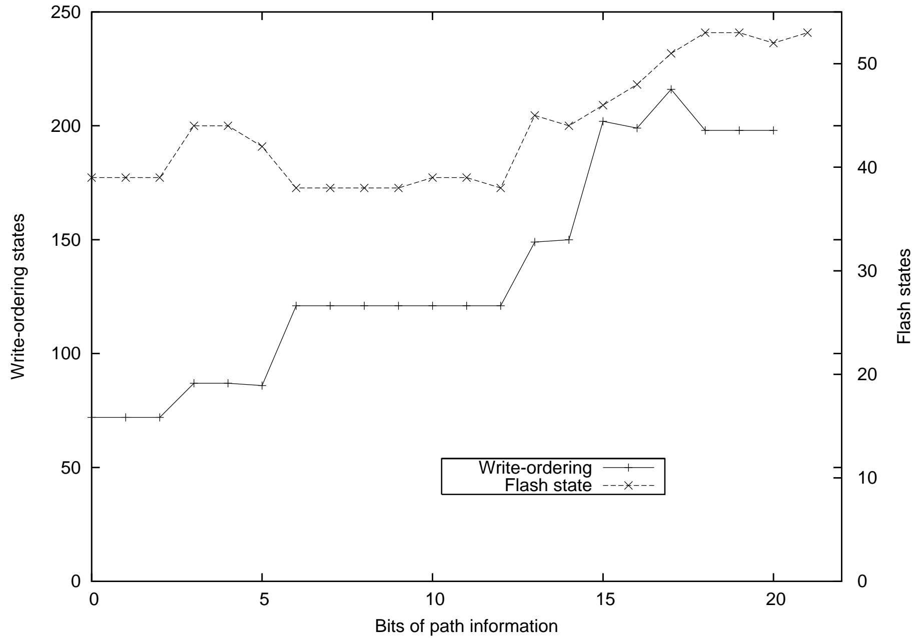

Fig. 5: Preliminary reachability results

In other words, considering *only path information* when determining whether to backtrack was almost as effective (in covering the abstract state space) as explicitly basing the search on abstract coverage[5].

Calculating coverage of another coarse abstraction (the ordering of writes to locations) showed even better results — in the two configurations we examined, the path-based approach quickly improved on matching the abstraction in question. With only 3 bits of path information, we were able to cover 87 abstract states, vs. 72 when matching on the abstract state. Figure 5 shows the general trend of increased coverage for both abstractions. Of course, given the high sensitivity of DFS to ordering of transitions [6], these results are at best suggestive. However, given the low overhead of the instrumentation and the difficulty of formulating useful (even unsound) abstractions for rich properties of systems with large state spaces, we believe this strategy merits further investigation.

4.6 Impact of Instrumentation on Model Checking Performance

Table 1 shows the overhead of instrumentation for model checking on our JPL examples. NVDS-1 and NVDS-2 designate different harness configurations. For the string library, two modes of verification were used — exhaustive model checking

[5] Coverage is incomplete in these cases because the abstraction is unsound and does not over-approximate successors.

	Uninstrumented		Instrumented				
Model Checking							
Program	Time	SPIN	Time	SPIN	Check	Slowdown	Type
NVDS-1	123.8	95%	137.1	88%	5.2%	10.5%	track
NVDS-1 (bitstate)	581.9	93%	621.3	86%	3.03%	6.8%	track
NVDS-2	437.4	93%	490.8	89.7%	2.08%	12.2%	pathBV(20)
Launchseq	97.6	99%	98.3	98%	0.06%	0.7%	track
`n_strncpy`	34.6	99.5%	34.9	99.4%	0.22%	0.86%	modifies
`n_strncat`	29.3	99.6%	29.4	99.5%	0.04%	0.34%	modifies
Random Testing							
Program	Time	Test	Time	Test	Check	Slowdown	Type
stringtest	202.9	80%	250.3	41.1%	24.4%	23.4%	modifies

All times in seconds. **SPIN/Test** are % time spent in core SPIN or in generating tests. **Check** is % time spent in `checkWrite` or in the `pathBV` update functions. Experiments performed on dual-core Xeon (3.2 GHz) with 8 GB of RAM, under Red Hat FC 4.

Table 1: Impact of instrumentation on performance

tests for each function and a random test system for the entire library. We report on modifies clause and path coverage instrumentations as representative — reset simulation instruments the same program points as modifies clause checking.

The slowdown for introducing instrumentation in no case exceeded 12.2%, whether instrumenting every global memory write or every branch, during model checking (**stringtest** is a random tester). For some of the programs, the overhead was below 1%. The overhead for modifies clause instrumentation is low enough that it can be used throughout the development of a SPIN harness to refine tracking statements, even when directly model checking flight software modules. The reason for the low (relative) overhead is clear: profiling shows that the time spent running instrumented code is trivial compared to the time spent hashing and storing states in the model checker. The percent of time executing `checkWrite` is typically an order of magnitude or more less than the percent spent executing the SPIN verifier. In unusual cases, it might happen that program execution dominates model checking state storage time, but we have never observed such a profile, even with complex modules such as NVDS. Note that the (relative) overhead for random testing (**stringtest**) of the string library is much higher than the other examples, as the generation of random tests is not computationally intensive. We also note that there was no overhead for checking `n_strlen` and `n_strncmp`, as CIL observes no global writes.

5 Related Work

Musuvathi et al. note that it should be possible to use a dynamic analysis, such as Purify [20] or StackGuard [5], in combination with CMC, a model checker that, like SPIN, executes C code [22]. In their experience, however, the overhead

of binary instrumentation is too high to be practical, which supports our decision to rely on source-to-source instrumentation [21].

In a sense, *any* analysis performed in an explicit-state model checker in which code is executed can be considered to be an instance of dynamic analysis during model checking. For example, the Java Pathfinder model checker [26] has been used to generate Daikon [7] traces to detect invariants [11]. Our contribution is to combine model checking with independent dynamic analysis, introduced via traditional source-to-source transformations. Our approach stands in contrast to the more common approach to applying a "dynamic" analysis during model checking, in which the model checker itself is extended to carry out the analysis, as with JPF [11]. We preserve a separation of concerns in which code may be instrumented *just as in testing or regular execution*, without substantial change to the model checker. Any analysis developed for model checking can also be used during normal testing or as a monitor for execution (as well as the reverse — instrumentation developed during testing can be applied while model checking). Perhaps more importantly, instrumented native code executes much faster than code executed in a virtual environment, such as JPF's JVM, and the techniques described in this paper should be applicable (with similar overhead) to any of the numerous model checkers featuring actual execution of C code, including CMC [22], VeriSoft [9], and CRunner [18].

6 Conclusions and Future Work

Applying independent dynamic analysis during model checking enables a large number of useful checks and measurements and saves effort by making it possible to use the same analyses during model checking, testing, and monitoring after deployment. Dynamic analysis can be at least as useful in execution-based model checking as it is in testing, especially given that relative overheads are typically much lower than in testing. We are exploring a number of other applications.

In runtime verification, certain program events are observed by a monitor, and temporal properties of program execution are checked [13]. We hope to reuse monitoring specifications developed for testing or deployed execution. We plan to integrate the RMOR runtime verification system [12] with SPIN to analyze properties of flight software. RMOR's instrumentation is already implemented as a CIL module, and uses only static structures for monitoring, which makes producing `c_track` statements to support monitoring relatively easy.

Another application is the inference, rather than checking, of properties. Daikon [7] infers state invariants of program execution and Perracotta [29, 28] infers temporal properties of program *paths*. The information necessary for these tools can be produced using our instrumentation approach. With Perracotta, it is critical to track path information and backtrack it during the search, to avoid inference of spurious properties. This idea also raises a question: when a tool examines traces offline, which traces should we generate during *model checking*? The most conservative approach would only produce traces at final states of the model. Given that an unbounded loop is the most common structure for a

verification harness this is not useful. The opposite extreme would be to produce a trace at each state reached. This would produce a very large (and highly redundant) set of traces. We suggest producing a trace every time the model checker backtracks due to reaching a final state or due to reaching an already visited state (but not after exploring all successors): this would produce one trace for each path by which the model checker reached any particular program state, even if that state was produced many times during model checking; no trace which is a prefix of another trace would be generated. Only empirical investigation can determine if such a strategy produces too many traces.

References

1. Thomas Ball. The concept of dynamic analysis. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 216–234, 1999.
2. Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
3. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
4. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.
5. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, 1998.
6. Matthew Dwyer, Suzette Person, and Sebastian G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Foundations of Software Engineering*, pages 92–104, 2006.
7. Michael Ernst, Jake Cockrell, William Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.
8. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.
9. Patrice Godefroid. Verisoft: a tool for the automatic analysis of concurrent software. In *Computer-Aided Verification*, pages 172–186, 1997.
10. Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, 2007.
11. Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
12. Klaus Havelund. RMOR Version 2.0 user manual. Kestrel Technology, California, USA, 2006.
13. Klaus Havelund and Allen Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments*, 2005.

14. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
15. Gerard J. Holzmann and Dragan Bosnacki. The design of a multi-core extension of the Spin model checker. In *IEEE Transactions on Software Engineering*, volume 33, pages 659–674, October 2007.
16. Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.
17. Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.
18. Daniel Kroening, Alex Groce, and Edmund M. Clarke. Counterexample guided abstraction refinement via program execution. In *International Conference on Formal Engineering Methods*, pages 224–238, 2004.
19. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, January 1995.
20. IBM Rational Software. Purify: Advanced runtime error checking for C/C++ developers. `http://www-306.ibm.com/software/awdtools/purify/`.
21. Madanlal Musuvathi. Email communications. 2007.
22. Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.
23. George Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.
24. Nicolas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.
25. Various. A collection of NAND Flash application notes, whitepapers and articles. Available at `http://www.data-io.com/NAND /NANDApplicationNotes.asp`.
26. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
27. Jeannette M. Wing. A two-tiered approach to specifying programs, 1983.
28. Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 23–28, 2004.
29. Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *International Conference on Software Engineering*, pages 282–291, 2006.