Model Checking Machine Code with the GNU Debugger

Eric Mercer and Michael Jones

Department of Computer Science
Brigham Young University
Provo, Utah, USA

Abstract. Embedded software verification is an important verification problem that requires the ability to reason about the timed semantics of concurrent behaviors at a low level of atomicity. The level of atomicity is the smallest execution block (such as a machine instruction or a C instruction) that cannot be split by an interrupt. Combining a cycle-accurate debugger with model checking algorithms provides an accurate model of software execution at the machine-code level while supporting concurrency and allowing abstractions to manage state explosion. We report on the design and implementation of such a model checker using the GNU debugger (gdb) with different processor backends. A significant feature of the resulting tool is that we can adjust the level of atomicity during the model checking run to reduce state explosion while focusing on behaviors that are likely to generate an error.

1 Introduction

Embedded software for small devices forms an important and unique verification problem. Embedded systems pervade many aspects of society and their complexity is growing quickly with processing power. If processor design continues to follow Moore's law, then current test strategies will not be able to sufficiently validate safety and capital critical embedded systems.

The principle challenge to embedded software verification is concurrency. Without fine-grain control of scheduling decisions in the operating system, it is not possible to explore the behaviors of concurrent interactions with typical debugging tools. A debugger is a familiar framework for software testing that closely reflects the behavior of the actual system because it is either running directly on the target hardware or on a high-fidelity back-end simulator. When running on native hardware, the debugger often uses hardware registers and traps, when necessary, to control program flow without significantly altering run-time behavior. Recent work in debugger and virtual machine technology implements reverse execution to step backward through code [21]. Although a debugger provides several mechanisms to control program execution and alter program state, it does not provide mechanisms to adequately explore concurrent interactions. As such, a debugger is not sufficient to validate embedded software with concurrency through to threads, processes, or interrupts.

A model checker is well suited to the systematic exploration of concurrent behaviors. Several techniques for software model checking are actively being pursued. The most common approach applies conservative abstractions to the high-level programming language [8, 2]. If no errors are found, then the program under test is error-free (relative to the given specification, of course). Counterexamples may be either infeasible or feasible. Infeasible counterexamples are used to iteratively refine the abstraction and feasible counterexamples are returned to the user. This approach is particularly successful in verifying control intensive code in which conditional expressions do not depend on extensively manipulated data values. Another approach applies bounded model checking to C programs and can verify buffer overflows, pointer safety, user defined assertions, and language consistencies [4, 5]. Other approaches translate the software under test into the formally defined input language of an existing model checker [17, 19, 1]. Language extensions are sometimes needed to facilitate the translation since the language semantics are not always directly supported by the existing framework [6]. A recent approach uses symbolic execution to verify properties of algorithms [16].

Each of the preceding approaches assume that the high-level language constructs are atomic operations. This assumption is adequate for the class of software properties verified in the various tools. In this work, we are interested in a concurrency model that more accurately matches the behavior of programs running on a given target processor. In most instruction set architectures, interrupts, due to concurrency or external inputs, can be taken between machine instructions and many C instructions are implemented with more than one machine instruction.

There are two approaches to software model checking that are directly pertinent to reasoning at a finer-grained level of concurrency. The first approach model checks the actual software implementation by instrumenting either a simulator or the virtual machine for the target architecture [22, 12]. This approach retains a high-fidelity model of the target execution platform. And, in the case of a Java virtual machine, there is low-level control of scheduling decisions. The second approach directly instruments the machine-code of the program and runs an analysis at speed on the native hardware [13, 15, 7]. Testing at speed can boost performance in state generation, and the overhead for instrumentation has been shown to be acceptable in large programs [20, 14]; however, timing information is skewed from the instrumentation. The work in [7] actually requires the source code to be annotated with model checking hooks for the instrumentation, and in some instances, access to the actual source code is not possible. These low-level approaches tie a tool directly to a target architecture or language, and the lack of abstraction leads to state explosion in the search space.

The work in this paper is based on this approach to software model checking but seeks to improve the process by interfacing with a standard debugger rather than working through a virtual machine or instrumenting the code under test. The central contribution is a better understanding of the challenges and opportunities of model checking machine code using a debugger. This understanding

is based on our extension of the GNU debugger (gdb) which supports model checking for a variety of target processors at a variety of dynamically tunable atomicity levels. A debugger provides an accurate model of software execution at the machine-code level while allowing abstractions to manage state explosion. Furthermore, working at the machine-code level through a modular debugger decouples the model checker from a particular high-level language and target architecture. Note that the debugger does not solve the state explosion problem directly; rather, it provides mechanisms that can be used to mitigate state explosion by altering the atomic step level of the model checker.

2 An Example

We illustrate the advantages of working at the machine-code level with a program, called SSE, that contains a simple serialization error. The error is manifest by a data inconsistency. Although the SSE program is somewhat naive, it illustrates the kinds of errors that can only be found when reasoning about concurrency at the machine-code level. In practice, more complex errors similar to the one in SSE arise when provably mutual exclusion techniques are either used or implemented incorrectly.

Figure 1 contains C and machine code versions of SSE. The machine code is a simplified version of the code generated by the GNU C compiler (gcc) for the Motorola 68hc11 processor. Simplifications are made strictly for readability in the figure. The analyzed code is the unmodified gcc output. The `while` loop in the C program contains an `if` statement that compares the readings of two sensors. If the readings are not equal, then an alarm is activated. The sensor readings are updated periodically by an interrupt handler (not shown) that copies readings from two input ports into the variables `reading[0]` and `reading[1]`.

In the assembly code for SSE, which is shown on the right side of Figure 1, the guard in the `if` statement is implemented with three instructions. The first instruction loads `reading[0]` (located at address 0x108e) into register D. The second instruction compares the contents of register D with `reading[1]` (located at address 0x1090). The third instruction branches past the alarm activation code if the values are equal. If the interrupt which updates `reading[0]` and `reading[1]` happens between the load and compare instructions, then the alarm

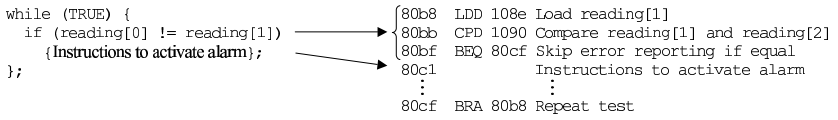


Fig. 1. The SSE example expressed in both C and assembly code. The guard in the `if` statement is actually implemented with three assembly instructions. Assembly code generated by the GNU C compiler for the Motorola 68hc11 processor.

may be incorrectly activated. The alarm may be incorrectly activated because one reading is stored in a register and the other in memory when the interrupt updates the contents of both in memory. Even if both readings are changed to the same value in memory, the now stale value in the register will be different. This particular interleaving is unreachable if the comparison in the guard in the `if` statement is modeled as an atomic comparison.

Of course, the serialization error in SSE can be eliminated by providing mutually exclusive access to the `reading` variables or by performing the check in the interrupt handler rather than in the busy-wait loop. If a lock is used to resolve the issue, then the lock can be implemented using any of a number of mutual exclusion algorithms that commonly appear as case studies in the model checking literature. The central verification issue addressed in this paper is not the correctness of mutual exclusion algorithms in general but the issue of whether or not a mutual exclusion algorithm was correctly implemented and used.

The SSE example can also be used to demonstrate the utility of including time in the processor execution model. The data inconsistency error in SSE can be eliminated by carefully scheduling the interrupts to occur only at safe locations between specific pairs of instructions. A fragment of machine code that preserves mutual exclusion using timing is shown in Figure 2. In the 68hc11, periodic interrupts are scheduled by writing a 16-bit value to a special timer "register" (two bytes stored at memory location 0x101c in this case) and setting a bit in a control register to enable the real-time interrupt. The interrupt is triggered when the free running counter is equal to the value written in the timer register. The free running counter is incremented by one in every clock cycle. The interrupt is serviced at the next instruction boundary after its corresponding interrupt flag is raised. The interrupt service routine clears the interrupt flag and schedules the next interrupt by writing a new value into the timer register. The new value is typically calculated by adding a fixed offset to the present value of the free running counter. Later, we will compare the model checking results for the timed version of SSE with the behavior of the same program on the target hardware. In all cases, the predicted behavior precisely matches the actual behavior.

The execution of the guard and body of the `while` loop in SSE requires 22 clock cycles. Starting at the instruction on line 0x8053, which sets the next time-out value, the interrupt handler requires another 115 clock cycles before it returns control back to the interrupt point in the `while` loop. This timing relationship is shown in Figure 2. If the interrupt that update `reading[0]` and `reading[1]` occurs with a period of $d = 22x + 115$ cycles (for values of x such that d is less than $2^{16} - 1$), then the update interrupt will alternate between the instructions at 0x80bf and 0x80cf. These are "safe" locations in which to update the `reading` variables. Another important aspect of this example is the amount of time that passes between the scheduling of the first interrupt and entry in the `while` loop. This delay fixed the location of the first interrupt in the `while` loop but is omitted from Figure 2 for clarity.

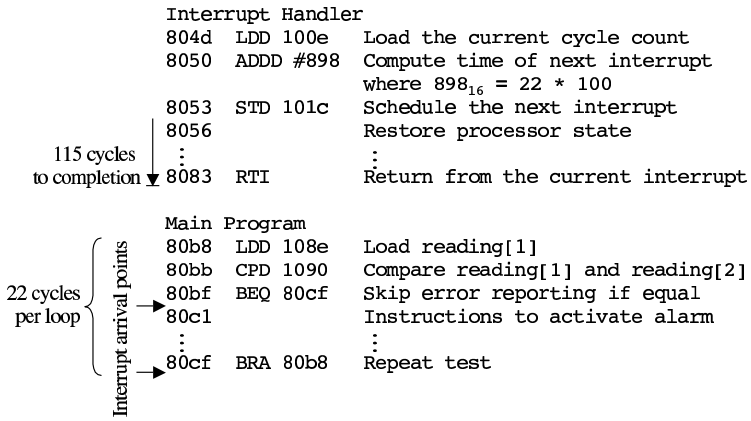


Fig. 2. Using timing to avoid the serialization error in the SSE example.

In practice, this approach to mutual exclusion is advantageous because it does not require locking protocols (which can degrade important performance metrics, such as response time). This approach is difficult to implement, without automated verification support, because it is conceptually difficult to correctly reason about timed concurrent behavior. A model checker that allows reasoning about timed concurrent behavior may extend design capacity by providing automated support for mutual exclusion that depends on timing. The goal of this work is to develop models and techniques to verify these types of systems and properties. Rather than create a processor model in an existing input language for a model checking tool, however, we are going to use the actual target processor hardware through a debugger.

3 State Enumeration by Debugger

The state enumeration process is inspired by, and closely resembles, state generation in the JPF2 and StEAM tools in that they instrument virtual machines to perform model checking tasks [22, 12]. In this case, the machine is real, not virtual, and the interface to the machine (or simulator) is a debugger. There are two principle tasks in state enumeration for explicit state model checking that the debugger needs to support: backtracking and resolving nondeterminism. In this section, we discuss the architecture of the resulting tools and how we deal with each of these tasks.

The general architecture for the model checker is shown in Figure 3(a). Roughly speaking, there are five major components to the architecture: the state, hash-table, processor, environment, and search models. We use the term "model" to indicate a generic representation of a particular component. The interface to

each model is designed to allow flexibility in a manner similar to that of the Bogor framework [19].

3.1 State and Hash-table Models

The state model represents the complete state of the processor and environment. It interfaces with the hash-table model using a *linearize* function which converts the state into an array of bytes. The state model is processor dependent; although, a generic configurable state model is provided that features commonly found in microprocessors including general purpose registers, control registers, counters and memory. The interface exported by this model can be redefined to meet the verification requirements of a given program or target processor. For example, a common specialization implements redefines the treatment of read-only registers.

The hash-table model interfaces with the state model through the one-way linearize function which, as mentioned previously, converts the state into an array of bytes for storage. This split between the state model and the hash-table model decouples the representation of the state vector in state generation and storage for duplicate detection. Such a split simplifies the implementation of different hash table architectures and storage disciplines. For example, super-trace (bit-state hashing) and hash-compaction can be implemented in the hash-table model without affecting the state model. The current implementation supports a collapse compression option [11, 9].

Although the state and hash-table models simplify the implementation of symmetry and partial order reductions, the implementation of certain symmetry and partial order reductions are more difficult if the operating system is part of the software artifact under test. If the operating system is *not* part of the software artifact, then standard implementation techniques that rely on access to operating system data structures can be applied. If, however, the operating system is included in the software under test, then it is more challenging to recognize symmetries because the necessary operating system datastructures are scattered blindly throughout memory in the state model. A specialization is required to extract the thread and heap information from the state model. This same argument also holds when discussing partial order reductions. Independence and visibility are tied to actions common between threads; thus, an understanding of threads must be extracted from the raw data. Debugging hooks in the machine code can facilitate access to this data but extra work is required.

3.2 Processor Model

The processor model is the execution framework for the target architecture. In the current implementation, we use version 6.1 of the GNU debugger (gdb) for the execution framework. This version of the debugger is easily reconfigured for cross-platform development using a collection of freely available back-end simulators. The actual input to the simulator, and hence the model checker, is a raw binary file in either elf or a.out format. The binary file may or may not

include debugging hooks to relate the machine-code to the high-level language from which it was compiled. The debugger can relate properly annotated machine code to a variety of high-level languages such as C, C++, Fortran and Ada.

The creation of a processor model is a pivotal point in the decision to create a new model checker rather than write processor models for use in another model checker, such as SPIN or Bandera. The central issue is the amount of work required to both create an accurate model of a processor architecture and implement the debugging features found in a debugger but not in any model checker.

The key feature found in a debugger, but not in a model checker, is the ability to supporting state generation at different levels of atomicity that can be changed during verification. Doing so in an existing model checker would require a significant rewrite. Currently, the actual steps the machine takes to update counters, process interrupts, etc. are invisible to the model checker. Moreover, we only record states at debugger break points, and these points can be defined in a variety of ways. The step level can be machine-code, high-level language, branch-point [3], or a mixed mode and all steps can be made conditional on run-time data values. Stepping at the branch-point level stops the debugger at points of nondeterminism that require an environment response. The mixed mode operates in any of the three levels and can be used to force the debugger to continue program execution until the program state satisfies a break point. This is useful in executing the program across system calls or program states that are of no interest to the property being verified. Stepping through library calls allows one to accurately determine the effect of a library call on the property under test.

Modeling the processor in an existing framework is itself a challenge because some processor behaviors are neither simply described nor simply implemented. Presumably, leveraging the effort expended to create an accurate simulator rather than writing a new one from scratch frees one to focus on other issues. Some of the more difficult processor functions include interrupt priority resolution, interrupt register register stacking and control register updates. Other aspects of processor execution, such as instruction interpretation are straightforward if not monotonous.

The vgdb debugger interface to the processor model in Figure 3(a) provides the backtracking facility necessary for explicit state enumeration . The interface takes a state model and loads it into the processor through the debugger. The debugger then turns control over to the target program which starts execution at the program counter in the loaded state. The debugger either steps at the machine-code level, the high-level language level, or until it runs to a breakpoint depending on the search model and user configuration. When the debugger stops, the model checker reads the resulting state from the debugger into the state model. Only modified parts of the state are updated in the state model. This saves time but still requires scanning the entire contents of memory in the debugger. This process can be further optimized with a map that identifies portions of memory that are either read-only, unaffected by the program under

test, or simply out of bounds. This information is given to the model checker at runtime along with the program to verify and its properties.

The management of read-only and clear-on-write registers is a challenge in model checking with a debugger. The free running counter used to track the passage of time in the m68hc11 processor is an an example such a read only register. This register can only be set at boot time or when the processor is in test mode. The register that marks interrupt arrives in the m68hc11 is an example of a clear-on-write register.

When using a simulator, read-only and special control registers can be made arbitrary writable by suitably modifying the simulator. GNU gdb is well suited to this because it provides call-backs to implement all of the debugger functions in a simulator including a special interface to send commands directly to a simulator. For example, we modified the back-end simulator for the m68hc11 to include a command that puts the simulator in a mode that bypasses the write logic for special control registers such as those used for interrupt flags. When writing to these registers, rather than clearing the flags to acknowledge the interrupts, the simulator sets the flags.

Writing to read-only and special control registers can be simulated in hardware by carefully manipulating the state model. For example, as mentioned previously, the free running counter in the m68hc11 is read-only, but we need to control this register because it affects the firing of real-time interrupts. To address this issue, the state model is specialized to store the difference between the current value of a timer register and the free running counter rather than the actual value of the timer register. When a state is loaded in and out of the debugger, the real-time interrupts are set to the current value of the real-time counter plus the difference stored in the state model. Similarly, when we read a state out of the debugger, we store the difference between the scheduled interrupt time and the current value of the real-time counter. Using this method, we are able to match real-time hardware interrupt behavior for the m68hc11. Another example relates to backtracking to states that have pending interrupts. Interrupt flags on the m68hc11 cannot be set. They can only be cleared. As mentioned earlier, we can modify the simulator to let us set the flags, only this does not work when running on the native hardware. To accommodate this, we create a state that causes the actual interrupt to fire on the next machine-instruction. This can be accomplished by carefully setting the real-time interrupts or toggling the external interrupt pin from the model checker through the serial or parallel port. Using this interface, the debugger can start execution from any arbitrary state in the program.

3.3 Environment Model

The environment model in Figure 3(a) closes the program under test by handling nondeterminism in a systematic way, checking invariants and configuring program dependent properties of the state. The environment model is implemented in C++ by the user, and must be compiled into the model checker to create

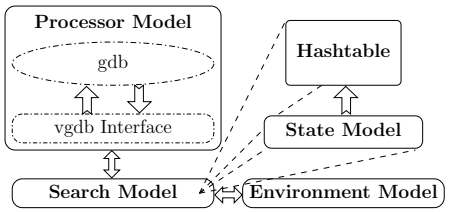
an executable that is specific to the program being verified.[1] More specifically, the environment provides a set of points to the model checker that represent either locations where an environment response is needed or locations where a property invariant needs to be checked. It is important to note that doing so does not require the source code for the program under test because these locations are instruction addresses. Aside from the controls used by the debugger, the program runs unaltered in the model checker. Each environment response transforms the state model appropriately and returns the updated state to the model checker. The invariant checks are predicates defined over the variables in the state model which can include information specific to the environment.

Environment specific state information can be (and often must be) stored in the state model. For example, modeling thread scheduling in the environment requires environment specific state. When model checking a multi-threaded program, it is possible to model check directly with the operating system, or it is also possible to abstract the operating system into the environment. If the operating system is abstracted, then the environment adds data to the state model to represent thread information. Instruction indices where we want to consider a possible scheduling operation are listed as points of nondeterminism in the environment model. The debugger stops at these indices, and the environment systematically generates states which explore the effects of different scheduling choices. Rather than providing a list of scheduling points, one could also use real-time interrupts in the target processor to implement round-robin scheduling. The interrupt handler can either implement a deterministic scheduling scheme or allow nondeterminism as before. Finally, the environment sets program specific state model properties. These properties might include read-only memory locations, track locations, match locations, and data abstractions.

3.4 Search Model

The final component for state enumeration is the search model that directs the traversal of the state space. Figure 3(b) shows pseudo-code for a breadth-first search model. The search model itself is an interface to the debugger which facilitate other search strategies. The breadth-first search example illustrates both the basic sequence of operations that might occur in a state enumeration strategy, and the interactions might occur between various components of the system. In Figure 3(b), HT is the hash-table, EM is the environment model, and PM is the processor model. The breadth-first search does not use undo information to backtrack. Instead, the search maintains a queue, Q, of frontier states to be expanded. After a state is dequeued on line 4, it is sent to the environment for possible invariants checking and nondeterministic responses. If the environment does not have a response for the given state, then it is returned unaltered. Each environment response is then sent to the processor model where

[1] Most of the model checker options are configured at runtime on the command line and compiled into a final executable in a manner very similar to that used in SPIN and Mur-ϕ.



(a)

Algorithm: bfs(s_o, PM, EM)

```
 1: init(Q, s_o)
 2: HT = {linearize(s_o)}
 3: while |Q| ≠ 0 do
 4:    s = dequeue(Q)
 5:    for all s' ∈ EM(s) do
 6:       s' = PM(s')
 7:       if linearize(s') ∉ HT then
 8:          if isViolation(s') then
 9:             reportTrace(s')
10:             return false
11:          HT = HT ∪ {linearize(s')}
12:          enqueue(Q, s')
13: return true
```

(b)

Fig. 3. The general architecture for model checking with a debugger and a search algorithm. (a) The general architecture showing the model checker interface to the debugger. (b) The breadth-first search algorithm using the processor model (PM), environment model (EM) and hash-table (HT).

it is loaded into the debugger, and the debugger begins execution. When the debugger stops, the new state is read, linearized, and sent to the hash-table for a membership check. If the new state is not a member of the hash table, then default properties are checked in line 8. These properties include stack overflow, read-only violations, and any properties general to most software programs.

The remainder of the pseudo-code in Figure 3(b) proceeds by updating the hash-table and adding the new state to the queue. The tool currently includes implementations of depth-first and guided-search in addition to breadth-first search.

4 Modeling Software

Model checking software requires extra care to properly handle functions, pointers, interrupts and external libraries. Model checking at the machine-code level simplifies the inclusion of functions, pointers, interrupts and libraries. This is primarily because the machine-code model must include all of information needed to handle each of these unique properties of software. This is both a blessing and a curse because the necessity of including such information further intensifies the state explosion problem. On the other hand, software model checking at the source-language level is somewhat more complicated because it must preserve the illusions about variables, function calls and program flow that provided to the programmer in a high-level language. These illusions include the notions that variables that range over all of the naturals (or reals) and that call stacks that can be arbitrarily deep. While reasoning about variables with infinite ranges is often simpler than reasoning about finitely ranged models, doing so would

destroy the accuracy of the resulting analysis. The challenge is to retain the accuracy while approaching the efficiency of infinite domain techniques.

Function calls are simplified because the calling context of every function is stored on the stack, which is part of the state model. There is a problem with recursive functions in which the depth of the stack is, theoretically, unbounded. In practice though, the stack is not unbounded and excessive recursive function calls will eventually lead to stack overflow. In some verification settings, particularly for embedded software, determining a bound for the stack size and detecting such stack overflows is itself a significant problem [18]. The depth bound on function calls and recursion using a debugger is the same as it is in hardware.

Pointers can be handled in machine-code models because the state model is simply the contents of memory rather than a logical model of memory that requires alias analysis. The problem with pointers, for high-level language models, is that they can reference arbitrary memory locations and this is difficult to model with a state vector that contains variables and their values (because the value of a pointer variable is an address of the value of interest rather than the value itself). A symptom of this problem is that when updating one variable's value, its not clear if any other pointer variables alias the same location and should also be updated. Working at the machine-code level eliminates this problem because there are no variables. There are only addresses and values in a large array. Essentially, we have the whole contents of memory in the state vector so dereferencing a pointer and resolving aliasing issues is trivial during model checking. The new problem is that the state vector contains *all* of the memory locations and this model may become large and difficult to update.

It should be mentioned that dynamic memory allocation, using a C-command like malloc, is also trivially modeled at the machine-code level. If the operating system is part of the state model, then the new memory is deterministically allocated according to the scheme implemented by the operating system. Otherwise, the environment model implements a possibly nondeterministic memory allocation scheme that mimics or approximates the actual memory allocation scheme. If the memory allocation scheme is deterministic, then pointers pointing to allocated memory can be compared across model checking runs. As with symmetry or partial order reductions, the implementation of symmetry reductions related to dynamic memory allocation can be implemented if care is taken to extract the appropriate information from either the environment or state model.

Interrupts allow program flow to transfer to an interrupt handler between any two instructions for which interrupts are enabled. Modeling interrupts in high-level language execution is somewhat awkward because an interrupt may appear during multiple machine instructions that implement a single high-level instruction. In addition, modeling interrupts requires an accurate model of the interrupt timing and priority scheme of the target processor. This is impossible to do using on the definition of a high-level language like C. At the machine-code level, an interrupt simply looks like another function call that is governed by the processor model rather than the program control flow. Using an accurate

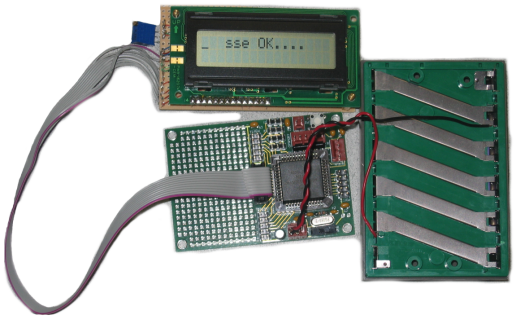


Fig. 4. A Motorolla 68hc11 processor used to validate Estes verification results for the tSSE problem.

simulator, or even the target processor itself, provides a good model of interrupt behavior with no additional effort.

Library function calls can, and indeed must be, be handled by either the environment model or as part of the state model. If library functions are included in the state model, then the environment model can be written to step over library function calls so that the effects of the function call are computed but the states reached in the function call are not stored or checked. Some function calls in some situations may be of interest. In these cases, conditional breakpoints can be set to step into function calls and include their behavior in the verification run when needed.

5 Results

The first results use the SSE program to motivate the need to use a fine-grained concurrency model and validate the accuracy of the model checker relative to the actual hardware. The program in Figure 1 does not specify when interrupts may or may not fire so we conservatively assume that they can fire between any pair of C or machine code instructions depending on the verification step level. As expected, if the verification is performed at the C language level, then the alarm always correctly activated because we do not allow interrupts inside the guard of the `if` statement. If the verification is performed at the machine-code level, then the alarm is incorrectly activated.

The SSE program in Figure 2 uses real-time interrupts to correct the incorrect activation of the alarm. Figure 4 shows a hardware implementation of the SSE system with the carefully timed interrupts on an m68hc11 chip. The verification is configured to step-over functions that update the LCD register. Doing so results in a significant savings in time and space because the code to write to the LCD is implemented using while-loops that require 2^{16} iterations times. We

Table 1. Results for the Motorola m68hc11 processor.

	C			Mixed		Machine code		
Program	lines	time	states	time	states	lines	time	states
Hyman	80	0.5s	387	2.2s	2005	255	5.9s	6948
Peterson	80	1.2s	655	48s	19423	4443	45s	31772
Dining Philosopher	295	0.8s	1520	NA	NA	595	2.8s	7722

also instrument the SSE code to count the number and location of the interrupts, as well as the interrupt sequence. The results from the debugger tool and the actual hardware match exactly on all points.

The next set of results simply demonstrate some applications of the resulting model checking. The results are from implementations of the Hyman [10] and Peterson mutual exclusion algorithms and a classic dining philosopher algorithm. The Hyman and dining philosopher algorithms both have errors that can be detected at the high-language level while the Peterson algorithm is correct. The results from a Pentium III 1.8 GHz processor with 1 GB of RAM are shown in Table 1. The machine code is again compiled for an m68hc11 chip with the debugger connected to a back-end simulator. We show results for C, machine code, and mixed levels. The C level considers only interrupts at C instruction boundaries. The mixed level steps over unimportant code, such as the LCD output in the Petersons example, but allows interrupts at the machine-code instruction boundaries. For the results at the C and machine-code levels, the *lines* columns are the total number of lines of code from either the C language file or the file created from an object dump of the binary executable . The *time* columns contain the wall-clock time measured using the Unix time program which includes actual start up and shutdown overhead for the model checker and debugger. The *states* columns give the total number of states found using a depth-first search when the search either finds an error or exhausts the state space (in the case of Petersons algorithm).

In Table 1, we inserted the ideal switchings between assembly and C code by hand. It is not always easy or feasible to construct the switching protocol by hand. Future work will explore heuristic methods for switching between step-levels depending on the property under test.

6 Conclusion

Model checking at the machine-code level using a debugger results in an accurate, timed model of the software under test. As expected, allowing concurrency at the machine-code level allows the detection of errors that are missed at the C-instruction level and further compounds the state explosion problem. Switching between levels of atomicity on-the-fly based on conditions evaluated at run-time allows one to focus verification effort on (and to contain state explosion within)

specific regions of specific execution paths in the software under test. Modeling software at the machine-code level simplifies handling some problems unique to software model checking such as pointers, function calls, interrupts and library functions because all of the information needed to resolve such issues is included in the state model. This also exacerbates the state explosion problem.

Future work on model checking machine code with a debugger focuses on methods for containing the state explosion problem. A variety of state analysis techniques can be applied to machine code to simplify model checking. Static analysis of machine code is difficult because variables do not always have well defined types and scopes. Since we have a fully executable state vector during model checking, we can pause a model checking run to redo the parts of the initial static analysis using the concrete information in the state vector. For example, a dead variable analysis can be performed at every breakpoint (or some other suitable frequency) using the contents of memory to disambiguate conditions and function call return addresses. Such a refinement of static analysis results would be useful in dead variable analysis and estimating the distance to an error or assertion state. Dead variable analysis is useful in reducing the size of the state vector and distance estimation is useful in heuristics that direct the search to likely error states. Finally, the incorporation of symbolic techniques using BDD or SAT algorithms using a variant of the track and match methodology may provide a further increase in model checking capacity.

References

1. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
2. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264, London, UK, 2001. Springer-Verlag.
3. G. Behrmann, K.G. Larsen, and R. Pelánek. To store or not to store. In *Proc. Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.
4. E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.
5. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
6. R. Joshi G. J. Holzmann. Model-driven software verification. In S. Graf and L. Mounier, editors, *11th International SPIN Workshop*, number 2989 in LNCS, pages 76–91, Barcelona, Spain, April 2004. Springer.
7. P. Godefroid. Software model checking: The verisoft approach. Technical report, Bell Laboratories, Lucent Technologies, 2003.
8. T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

9. G.J. Holzmann. State compression in spin. In *Third Spin Workshop*, Twente University, The Netherlands, April 1997.
10. H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, 1966.
11. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. *Lecture Notes in Computer Science*, 2057, 2001.
12. T. Mehler and S. Edelkamp. Directed error detection in C++ with the assembley-level model checker StEAM. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, Barcelona, Spain, March 2004. Springer.
13. N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, September 2004.
14. N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004.
15. N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.
16. C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, Barcelona, Spain, March 2004. Springer.
17. J. Penix, W. Visser, C. Pasaranu, E. Engstrom, A. Larson, and N. Weininger. Verifying time partitioning in the DEOS scheduling kernel. In *22nd International Conference on Software Engineering (ICSE00)*. IEEE Press, 2000.
18. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT 2003)*, 2003.
19. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, 2003.
20. J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, April 2005.
21. Virtutech. Simics hindsight. http://www.virtutech.com/products/simics-hindsight.html, 2005.
22. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.