Formal Verification of Software Source Code Through Semi-Automatic Modeling

Cindy Eisner

IBM Haifa Research Laboratory

Abstract. We describe the experience of modeling and formally verifying a software cache algorithm using the model checker RuleBase. Contrary to prevailing wisdom, we used a highly detailed model created directly from the C code itself, rather than a high-level abstract model.

Keywords: software model checking, software verification, program verification, functional verification

1 Introduction

In recent years, the technique of formal verification known as model checking has gained wide acceptance as a powerful tool for hardware design, and has become an integral part of the verification process in IBM and other companies [17, 4, 28, 21, 42, 1, 49]. With the maturation of the method for hardware verification, the obvious question is whether the experience is transferable to software. A first step in the application of IBM's model checking tool RuleBase [7, 6] to software was described in [19]. In that project, RuleBase was applied to a highly abstracted model of RuleBase itself, in order to detect problems in the use of the garbage collection mechanism by the programmer. While the work described in [19] was successful, detecting eight bugs in a version of RuleBase under development, the high level of abstraction it used is not applicable to most real-life problems. This report describes a second, much more ambitious, step: the application of RuleBase to a concrete model of a piece of "hard to verify" software - a software cache algorithm.

The software verified is a distributed storage subsystem software application. In other words, it is the software that runs on hardware that connects one or more computers with one or more magnetic disks. It is a distributed application that runs on several nodes, providing full data availability after the failure of a single node, and at least partial service in the event of simultaneous failures. Like most storage subsystems, it contains a cache to speed up read and write operations. The algorithms used to implement the software cache are the subject of the work described in this report.

A software cache, like a hardware cache, holds a copy of the data stored elsewhere (on disk in the case of a software cache, in main memory in the case of a hardware cache). The copy can be either *clean*, which means that the data is identical to that on disk, or *dirty*, which means that the cached copy is different from (and newer than) that on disk. If the cached copy is dirty, then the cache algorithms must ensure that any read of the dirty address gets the newer (cached) copy, and that the new data is written back to disk before the address is discarded from the cache. Because the system is distributed, the cache algorithms must also provide the coordination between multiple nodes. In the case of multiple cached copies, all are identical. Because the system must ensure high availability, there is an additional functionality, which is to ensure that there are always at least two copies of dirty data; a second copy is used in the case of a failure of the node holding the first copy.

Since hardware cache protocols are classic candidates for formal verification [12, 36, 17, 21], formally verifying a software cache algorithm seemed like a natural next step in the effort to transfer the knowledge and experience of formal verification from hardware to software. The original goal was modest: to verify a high-level manually written model of the software cache algorithms, in the same way that early formal verification work on hardware used manually created models of the real thing. In this way, the thinking went, the many problems of software modeling not solved by the original effort in [19] (for instance, the modeling of pointers and complex data types) could be avoided, while still providing real value. The intention was that only after the algorithms were well understood would an attempt be made to verify the software directly.

However, it soon became apparent that this strategy would not work; it would not be possible to build manually a good model of the software algorithms to be verified. The reason for this is rooted in the fundamental difference between software and hardware. That difference is not necessarily what first comes to mind when considering the matter. The complex semantics of programming languages, including pointers, function calls, recursion, etc., are not a fundamental problem, despite the fact that hardware semantics are trivial by comparison. Neither is the fact that software is generally viewed as having infinite state, while the model checking algorithm used by RuleBase needs a finite state system. Rather, it is the fact that the control flow of software differs at such a basic level from that of hardware. In hardware, everything works in parallel, and this is reflected in the style of simple Hardware Description Languages, including EDL [24], the input language of RuleBase. In such languages, every statement is implicitly parallel to every other statement, and when sequential constructs, such as the EDL *process* exist, they are atomic. In software, however, control flows sequentially through a process, and when concurrency exists, it is between non-atomic entities. This difference in the granularity of concurrency is the difference that makes it hard to manually translate a software algorithm into the hardware modeling style required by EDL.

Because of the modeling problems described above, it was decided not to model the software directly in EDL, but rather to model in C, and to use the automatic translation to EDL described in [19]. The C model is very similar to the actual C code, to the extent that there is a one-to-one correspondence between a line of the model and a line of the original code. The main differences are a simplification of complex data types and a slight change to the mechanism for asynchronous callback. Both of these issues are explained in detail in Section 5.

The automatic translation solves the modeling problems that would be difficult to deal with manually. However, the translator of [19] had been used only once before, for a very particular purpose, and significant enhancements were required to make it useful on more general C code. The restricted translator of [19] and the more general translator developed for the purposes of this project are described in Section 5.

2 Comparison with related work

Many previous works have described the process of verifying high level models of software [10, 33, 34]. In this paper, we apply model checking to the source code itself, rather than to a hand coded high level model.

There is extensive previous work on the application of model checking to the source code of railway interlocking software [29, 37, 8, 9, 18, 20]. While technically a railway interlocking is a piece of software, the semantics of railway interlocking languages are extremely simple, to the extent that Sheeran and Stålmarck term interlockings *hardware-like* systems [45]. In this paper, we apply model checking to software written in the general purpose language C.

Godefroid [26, 27] describes VeriSoft, a tool for model checking concurrent software written in C or C++, and the successful verification of a 2500 line concurrent C program is noted. The focus of [26, 27] is the search algorithm, which performs a variety of explicit state space exploration. Stoller [46] takes an approach similar to that of [26, 27] for Java programs. In this paper, we do not modify the model checking algorithm. Rather, we use c2edl to translate C code into the input language of our model checker, and use the existing algorithms to verify certain useful properties of the program.

Demartini, Iosif and Sisto [15] describe the application of the SPIN model checker to Java multithreading applications. They describe the process of translating Java source code into PROMELA, the input language of SPIN. Their goal, like that of this paper, is to verify source code, using automatic abstraction techniques to get a simplified model. They demonstrate their technique on toy examples. Havelund and Pressburger [30] take an approach similar to [15] in the first generation of their tool Java PathFinder, but support more of the language, and note results for Java programs of up to 2000 lines of code. In both [15] and [30], the translation is complicated by the need to model the concurrency primitives of Java, while the method used by c2edl is free of those concerns. On the other hand, the translations of [15, 30] are in some ways simpler than that of c2edl, because the PROMELA language allows them to retain much more of the structure of the original program than does EDL. The issue of the ease of the translation process is not, however, the important difference between translating into PROMELA and translating into EDL. Rather, the important difference is that SPIN, the model checker that reads PROMELA, is an explicit state model checker, while RuleBase, the model checker that reads EDL, is a symbolic, or implicit state, model checker. A comparison of symbolic vs. explicit state model checking of the software system described in this paper can be found in [22].

Visser, Havelund, Brat and Park present the second generation of Java PathFinder in [48]. While the first generation translates Java source code into PROMELA, the second generation is a full-blown custom-made model checker for Java. In contrast, we have not developed a new model checking algorithm, but use modeling techniques to allow the application of an existing one.

Holzmann and Smith [32] present a method for extracting verification models from source code that results in a control-flow skeleton, using an abstraction process that is semi-automatic. They describe the results of an application of their method to commercial call processing software written in C, although they do not mention the size of this software. In [31], Holzmann describes another application of the method to a checkpoint management system. Again, the size of the software is not discussed. In contrast, we check a detailed, rather than abstract, model of the software.

Corbett et al [14] describe Bandera, a tool for automatic extraction of finite state models from Java source code. They perform user-guided abstractions based on reducing the cardinality of data sets, and provide a language for specifying additional abstractions. They translate Java to an intermediate language which is then translated to one of a number model checking languages. They demonstrate their method on a toy example, a threaded pipeline consisting of 60 lines of Java code. In contrast, we present results for a non-trivial application.

Esparza, Hansel, Rossmanith and Schwoon [23] describe model checking algorithms for pushdown automata. They take the radical approach of abstracting away all variable values, and are not limited to a finite stack. They give impressive results for randomly generated flow graphs (skeleton programs) of up to 20,000 lines. In contrast, the work described in this paper uses a highly detailed, as opposed to an abstract, model.

Finally, Ball and Rajamani [3] describe Bebop, a symbolic model checker for boolean programs. They have developed a specialized algorithm for model checking software, which appears to be limited to checking properties which are directly represented by the user as reachability queries. Like [23], they are not limited to finite state systems. In contrast, we use our existing model checker, and check properties expressed in temporal logic. Their approach to the semantic difficulties of software is to limit all variables to boolean values. They show results for a simple family of programs with increasingly deep levels of nested procedure calls, but limited non-determinism. In contrast, we choose to deal with more complicated real-life programs.

3 Two Early Decisions

Before describing the work that was done on this project, two early decisions should be discussed. The first is the decision to use RuleBase, a tool designed for verification of hardware, rather than building a tool specialized for software from scratch. The second is the decision to use a very detailed model of the cache algorithms, rather than a highly abstract high-level model.

3.1 The Decision to use RuleBase

Given the huge gap between the world of software and that of hardware described above, the decision to use RuleBase for software verification may seem strange. The explanation for this decision lies in the fact that it is only the modeling language EDL that is unsuitable for modeling software; the underlying algorithms used by RuleBase are as relevant to software as they are to hardware. To understand this, recall that software is merely an input to a piece of hardware. With RuleBase, we routinely model hardware and its inputs, so there is no fundamental reason that RuleBase should not be applicable to software. Indeed, a first attempt [19] proved the concept on a sizable piece of code, albeit on a highly abstracted model.

Not only is the application of RuleBase to software possible, there is a big incentive to make it work. Many person years of effort have gone into the tool since its birth in 1994, most of it to the core model checking algorithms which are as applicable to software as they are to hardware. In addition, the deep understanding of the underlying algorithms used by RuleBase that has been gained through years of application to hardware would take many additional years to duplicate on a completely new tool.

For these reasons, it was decided to further explore the use RuleBase for the verification of software, rather than to undertake the development of a brand new tool.

3.2 The Decision to Use a Detailed Model

Another early decision was that of using a detailed model, with a one-to-one correspondence between its lines of code and that of the actual software. There are several reasons that this approach, which goes against a lot of the conventional wisdom of the model checking community, was taken. First, modeling an algorithm in detail is easier than modeling the algorithm at an abstract level, especially when an implementation already exists. Writing the detailed C model of the cache algorithm entailed solving some highly isolated problems, while leaving most of the code untouched. The only structural changes made were a slight change to the mechanism of asynchronous callback, in order to eliminate the need for an additional thread, and a few simple changes to the way return codes are handled, in order to save state variables which are extremely expensive in model checking. Thus, writing the model entailed merely copying the original C code, using global replace to flatten structures and replace pointers with indices to pre-allocated arrays, and editing manually in a few places to change the structure. This process required only a basic understanding of the algorithm. In contrast, writing an abstract model requires detailed knowledge of the algorithm under verification, and is frequently a difficult process even then.

Another advantage of a detailed model is that it eliminates in large part the problem of knowing whether or not the results on the abstract model are relevant to the actual code. When a highly abstracted model is used, the problems of *false negatives* and *false positives* are usually acute. A false negative results when the abstraction process causes a specification to fail, not because the algorithm itself is buggy, but rather because the abstraction was performed too zealously. A false negative is discovered by examining the counter-example provided by the model checker, and requires recoding the abstraction to correct the erroneous behavior. The problem of false positives is more serious. This occurs when, as before, the abstraction does not accurately reflect the actual code. However, in a false positive, the abstraction obeys the specification, while the actual code does not. There is no way to detect a false positive without an added step in which it is proved that the abstraction is an accurate reflection of the software. This can be done, for instance, by proving a *simulation relation* [13, 38] between the abstraction and the code, or by developing the code from the abstraction through a process of *refinement* [2, 40, 39]. Unfortunately, this step is frequently as or more difficult than the model checking process itself. When a detailed model is used, neither problem is completely eliminated. However, because of the high similarity between the model and the actual code, neither is a serious obstacle.

Finally, the viability of a highly detailed model was shown in previous work, in which a complicated hardware cache protocol was verified with success [21]. In the formal design of the coherence unit of a multi-processor system, described in [21], 37 bugs were found in an early version of the design, and 50 bugs in the version that was sent to production.

For these reasons, it was decided to use a detailed model of the software cache, rather than to deal with the problems of a highly abstract one.

4 RuleBase

RuleBase is a formal verification tool that uses the technique of symbolic model checking in order to decide whether or not a design obeys its specification. A brief overview follows. A detailed tutorial on symbolic model checking can be found in [20].

In model checking [11, 43], a (finite) design is viewed as a Kripke structure (similar to a finite automaton, or a finite state machine), and the verification of the properties is achieved by traversing the structure. Consider for instance the Kripke structure shown in Figure 1. For this structure, we might be interested in verifying the following property:

If a request is made, it will be processed within three steps. (1)

We express this formally in the temporal logic CTL [11] as follows:

$$AG(request \rightarrow AX(process \vee AX(process \vee AXprocess))) \tag{2}$$

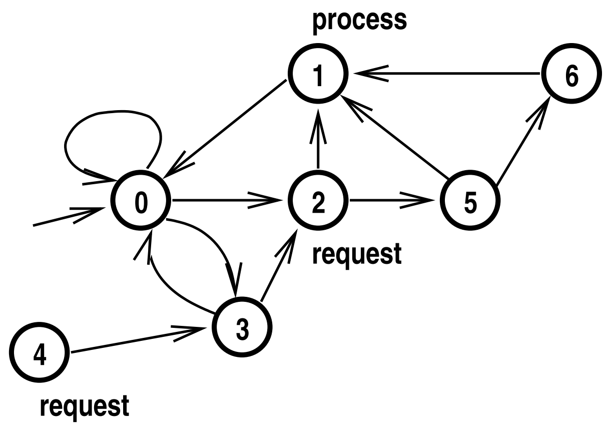


Fig. 1. Structure Request-Process

Formula 2 can be read as: "always globally (AG), if there is a request, then always on the step after that (AX), either process it, or always on the step after that (AX), process it, or, always on the step after that (AX), process it". The pre-condition of the formula, "if there is a request", holds only in states 2 and 4. By traversing the Kripke structure from those states using standard graph traversal algorithms, we can discover that from state 2, the property holds, and that from state 4, there are several paths which do not obey the requirement. One of these, for instance, is the infinite path $(4, 3, 0, 0, 0, \cdots)$. Thus, of the two relevant states, Property 2 holds only for state 2. Furthermore, by traversing the graph from the initial state 0, we can discover that the single state 4 which does not obey our property is not reachable, that is, it is not possible to get to state 4 from the initial state. Therefore, our structure satisfies the desired property.

Model checking an explicit representation of a Kripke structure like that shown in Figure 1 is fairly straightforward. However, building a Kripke structure for even a small real-life design is usually impossible. For instance, a piece of hardware with n latches or memory elements will have a Kripke structure with 2^n states. A piece of software with n boolean variables will have even more than 2^n states, as the program counter and stack must also be represented. For even relatively small values of n, this is prohibitive. The solution to this problem is the technique of symbolic model checking [36].

Symbolic model checking works by manipulating functions representing the structure rather than by building and traversing the structure itself. Since the effect of symbolic model checking is as if we had built and traversed the structure, we say that a symbolic model checker performs *implicit* state space exploration, while a non-symbolic model checker performs *explicit* state space exploration. An implicit representation of a Kripke structure is simply a description of the behavior of each bit of the system. For an n-bit system, the implicit representation is of size n, whereas the explicit representation, if built, would be of size 2^n.

The implicit model checker RuleBase takes three inputs: the system to be verified, a description of the environment (the legal behaviors of the inputs to the system under verification), and a description of the specification against which the system should be checked. For hardware, the system itself is usually written in one of two standard Hardware Description Languages (HDLs), Verilog or VHDL. The environment is described in EDL, the Environment Description Language of RuleBase[1]. The specification is described in Sugar [5, 25], the temporal logic used by RuleBase. These three inputs are compiled down by RuleBase into a model, comprised of the design itself and its environment, described in the language SMV [36], and its specification, described in CTL.

5 Modeling Software Using EDL

The first task in model checking software using RuleBase is to be able to support a standard software language such as C. The basic idea behind translating C into EDL[2] (so that RuleBase need not be modified at all in order to be applied to software) was presented in [19]. It is described in the next section in order to make this report readable as a standalone document.

5.1 Modeling Problems Solved Previously

This section describes how to express a program, including functions and recursion, as a set of next-state functions directly suitable for model checking. It is a recap of the process described in [19], which itself directly follows from [35, 13]. The process is quite simple, and is described by means of short examples.

Consider first the C function getmax() of Figure 2. We start by annotating the code with the value of the program counter (*pc*). We then restrict the integers *a*

```
    getmax (){
      int max, a;
0     a = max = 0;
1     do {
2       if (a > max)
3         max = a;
4       a = input();
5     } while(a);
6     return(max);
7   }
```

Fig. 2. Function getmax() in C

[1] The distinction between the language used to describe the design and that used to describe the environment is an artificial one. EDL is in fact a very simple HDL.

[2] Since C is the language used to describe the program under verification, it might seem more intuitive to translate C into Verilog or VHDL, which are used to describe hardware designs. However, as noted above, EDL is actually a very simple Hardware Description Language, so this choice is of minor importance.

and *max* to a finite range, say 0 through 3. If we interpret the call to $a = input()$ on line 4 as a non-deterministic assignment to the variable a, it is a simple process to rewrite getmax() in terms of next-state functions of the variables, as shown in Figure 3. With minor syntactic changes and the addition of state variable

```
next(a) = if pc=0 then 0
          else if pc=4 then {0,1,2,3}
          else a
next(max) = if pc=0 then 0
            else if pc=3 then a
            else max
next(pc) = if pc=0 then 1
           else if pc=1 then 2
           else if pc=2 then if a>max then 3 else 4
           else if pc=3 then 4
           else if pc=4 then 5
           else if pc=5 then if a then 1 else 6
           else if pc=6 then 7
           else if pc=7 then 7
```

Fig. 3. Function getmax() in terms of next-state functions

declarations, Figure 3 is a complete EDL program, and can be model checked using RuleBase.

The implementation itself is very simple. After parsing the source code, the program counter is allocated by traversing the parse tree. Generating the behavior of the program counter is then a matter of traversing the numbered parse tree a second time. Extending the translation to other kinds of branching and loop statements is straightforward. Handling functions, including recursive calls, is also fairly simple, and follows the standard method of a stack and stack pointer used by compilers: During the second traversal of the parse tree, information needed to generate propositions *somecall* (indicating a function call), *somereturn* (indicating the end of a function or a return statement), and *nextpcnocall* (indicates the return point to be pushed onto the stack for a function call) is gathered. Using these propositions, the behavior of the stack is described as shown in Figure 4. As in real life, the depth of the stack is finite (in

```
next(stackp) = if somecall then stackp_inc
               else if somereturn then stackp_dec
               else stackp
next(stack(stackp)) = if somecall then nextpcnocall
                      else stack(stackp)
stackp_inc = if stackp = max_stackp then stackp else stackp+1
stackp_dec = if stackp = 0 then stackp else stackp-1
```

Fig. 4. Standardized behavior of the stack

this case, limited to the value of *max_stackp*). Stack overflow is checked using a CTL formula. The next state value of the *pc* is then dependent on the propositions *somereturn* and *returntowhere*, as shown in Figure 5. The behavior of

```
next(pc) = if somereturn then returntowhere
           else ...
```

Fig. 5. Next state value of *pc* in presence of stack

returntowhere is itself a function of the stack, as shown in Figure 6.

```
returntowhere = case
                stackpminus1=0:stack_0;
                stackpminus1=1:stack_1;
                stackpminus1=2:stack_2;
                stackpminus1=3:stack_3;
                stackpminus1=4:stack_4;
                stackpminus1=5:stack_5;
                ...
```

Fig. 6. Modeling of *returntowhere* as a function of the stack

We have shown how to model a C program with simple data types, and function calls, including recursion. This process was completely automated in the tool c2edl as described in [19]. However, for the work described in this report, it was necessary to provide a more complete solution. This is the subject of the next sub-section.

5.2 Modeling Problems Solved by this Work

The software cache algorithm code contains many constructs not covered by the basic solution presented in the previous sub-section. Among these are: use of complex data types, such as structures and pointers, overloading of memory location names through the use of unions, complications of control flow due to multi-threading, including asynchronous completion of requests, and support for local variables and parameters. Some of these were solved automatically, through enhancements to the tool c2edl, and some were solved manually through changes to the C model. Each is discussed in detail below.

The term *pseudo-code* used below indicates the model. This is somewhat of a misnomer, since the model is perfectly legal C code. It is intended to indicate that the model is not quite the real thing.

Structures Structures were flattened into sets of individual variables. For instance, the structure *io* shown in Figure 7 is flattened into two variables: *io_qnode*

```
struct {
        snode node;
        void * callback;
} io;
```

Fig. 7. An example structure

and *io_callback*. Although this was done semi-manually by a global replace of references to *io.qnode* with *io_qnode*, it could have easily been automated.

Pointers Pointers were converted into integer indices into an array of variables of the types pointed to. For instance, the pointer $cache\_info$ shown in Figure 8 was modeled as an integer with range of 0 through MAXENTRY, where MAX-

```
cache_entry *cache_info ;
```

Fig. 8. An example pointer

ENTRY is configurable. Since $cache\_info$ is a pointer to a structure which was flattened as described above, the reference $cache\_info \rightarrow info.flags.pseudo$ appears in the pseudo-code as $cache\_info\_info\_flags\_pseudo[cache\_info]$, where $info\_flags\_pseudo$ is the flattening of the nested structures $info$ and $flags$ as described above, $cache\_info\_info\_flags\_pseudo$ is an array of the values of field *pseudo* for each pointer $cache\_info$ allocated, and $cache\_info$ is the integer index into that array. The conversion was done manually, although it could have easily been automated.

Unions The model does not contain any overloading of memory location names resulting from the use of unions in the actual C code. In some cases, this required no change to the C code, because some unions are used in such a way that a field is always accessed with either one name or the other. For instance, in the union shown in Figure 9 the field *pseudo* of structure $flags$ is never accessed as

```
union u {
    struct {
        unsigned pseudo:1;
        unsigned modified:1;
        unsigned local:1;
        unsigned reserved : 29;
    } flags;
    struct {
        unsigned char reserved1;
        unsigned char state;
        unsigned char substate;
    } state;
} info;
```

Fig. 9. An example union

field $reserved1$ of structure $state$.

Other unions are used in the C code in such a way that there is importance to the fact that a field has two or more names. For instance, the union shown in Figure 10 is used in such a way that field *write_owner.client_out_iob* is sometimes

```
union {
   struct {
       buffer io_request ;
       buffer owner_response;
       buffer output;

       bref io_request_iob;
       bref owner_response_iob;
       bref output_iob;

   } dir;
   struct {
       buffer io_request ;
       buffer reserved ;
       buffer client_out ;

       bref io_request_iob;
       bref reserved_iob;
       bref client_out_iob;
   } write_owner;
}
```

Fig. 10. Another example union

accessed as *dir.output_iob* in the C code. For these unions, the model differs from the C code in that only one of the two possible names is used.

This canceling out of name overloading resulting from unions was done manually, and was a painful and error prone process. Automating this step could have been easily automated, and in retrospect, should have been.

Multi-threading The solution presented above and in [19] assumes a single process or thread with a single program counter. In the case of multi-threading, the solution is the same, except that each thread must have its own program counter. Then, it is a simple matter to add a variable that acts as a guard to each thread, allowing only one to progress at any one time step. Giving the additional variable non-deterministic behavior ensures that every possible interleaving of threads is allowed. However, things are complicated by the fact that the threads existing in the model do not parallel those of the actual code.

Conceptually, the C code contains a thread per request (e.g. read or write). However, for performance reasons the C code contains only three threads, which manage the conceptual threads through a queueing system. In the model, each conceptual thread appears as an actual thread, and there is no representation of the queueing system. The main reason that this decision was taken is that modeling the algorithm in this way is much easier than modeling the queueing system. Since the model with one thread per request contains more behaviors than the actual C code (more interleavings of threaded code is possible than in the queueing system, which changes control from thread to thread at predefined points in the code), modeling the algorithm in this way cannot hide bugs (except for those in the queueing system itself). Furthermore, a real node may be comprised of a multi-processor, with one queueing system running on each. Thus, the additional interleavings of the model over the queueing system reflect additional real behaviors of the code on a multi-processor system.

One additional complication of the queueing system is that a conceptual thread can call another conceptual thread, by passing a pointer to its work buffer as a parameter. The translation process presented in Section 5 cannot deal with such behavior directly. What the model does is to add signals between threads. When the actual C code calls another thread, the model sends a signal to the thread instead, telling it to wake up at the point of invocation of the C code.

The solution described for multi-threading was coded manually.

Mutual Exclusion As a result of the multi-threading described in the previous sub-section, it was necessary to add support for mutex primitives. This was accomplished by modeling the behavior of the primitives in simple non-atomic pseudo-code, then restricting the behavior of the non-deterministic variable controlling the interleaving. Rather than completely non-deterministic behavior, this variable behaves as follows: if the *pc* of the currently enabled thread corresponds to a line of code inside one of the mutex primitives, the variable keeps its value. Otherwise, the next state of the variable is non-deterministic.

The solution described for mutual exclusion was coded manually.

Asynchronous Completion of Requests In the actual C code, requests for resources can be completed either *synchronously* or *asynchronously.* Synchronous completion occurs when the resource requested is immediately available, and simply results in the continued processing of the read or write request. When a requested resource is not immediately available, the resource allocation function returns a value of $WAIT$, and the conceptual thread is queued on a waiting list. When the requested resource becomes available, the conceptual thread is awakened. This is known as asynchronous completion. Asynchronous completion requires special handling in the model, to replace the queueing system of the actual C code. Asynchronous completion is modeled by sending a signal to the waiting process, in the same way that one thread calls another thread by sending it a signal.

The modeling of asynchronous completion was coded manually.

Local variables One of the characteristics of local variables is that a recursive function call gets its own copy (unless the variables are declared as static). The solution presented in [19] does not deal with this behavior, since the main application described in that work abstracted away all data values. Regarding the work described in this report, the model of the software cache algorithms contains no local variables which need this behavior to function correctly. Therefore, in order to save space and modeling effort, this feature of local variables was ignored.

Parameters Support for parameters is the one major feature that was added to the tool c2edl in the course of the work described herein. Support for parameters is complicated because of type problems. A compiler can pass parameters on the stack, using the entire width of the stack to store the value or address being passed. In model checking, however, saving bits is of primary importance, and passing parameters this way would be wasteful, because the number of bits needed to represent the program counter in the model is much greater than the number of bits needed to represent the variable with the largest range. Even if automatic reduction/abstraction techniques detected that parameters need less bits than the program counter, it is difficult to make use of this information in a model in which the same stack location may be used to store a parameter or a return value. Therefore, we allocate parameters on a dedicated stack called the *parameter stack*, which is not necessarily of the same width as the stack used to save the *pc*.

For the parameter stack, we must be able to represent all of the types. In theory, only n bits are needed, where n is the maximum number of bits needed to represent any one parameter. However, this would involve a lot of translation back and forth between encodings. In order to simplify the process, we model parameters using more bits than strictly needed, by declaring them to be of an enumerated type whose values are the union of the possible values of each parameter. For instance, if we have two functions with parameters in the model to be translated, and one can take the values 0 through 2, while the other can take the enumerated values $IDLE, STATE1, STATE2$, and $DONE$, then the parameter stack is modeled as a set of enumerated variables whole possible values are $\{0, 1, 2, IDLE, STATE1, STATE2, DONE\}$.

The parameter stack is modeled as a set of arrays, one for each stack depth. The array *param_stack*0, for instance, represents the top of the stack, while the array *param_stack*1 represents depth 1 of the stack. The array index indicates the parameter number. Thus, if the function with the largest number of parameters has n parameters, these arrays will be of dimension n. Figure 11 is the next-state representation of the parameter stack at depth i for parameter j.

```
next(param_stack_i(j)) = if i=stackp & somecall then params(j)
                            else param_stack_i(j)
```

Fig. 11. The parameter stack

6 The algorithm

Above we have sketched some details of the algorithm used by c2edl. This section gives an overview of the algorithm as a whole. In the next section, a complete example that uses the algorithm is presented. The steps are as follows:

1. Parse. During the process, allocate dummy *push call* statements, one before each call to a function which has been defined (for ease of use, calls to func-

tions which have not been defined are considered as noops, and a warning is printed. This allows properties of not-yet-complete software to be verified.).

2. Annotate the parse tree with program counter values for each statement. Only functions transitively called from the top-level function as indicated to c2edl will be numbered (that is, it is possible to use c2edl to build a model of other than main()). During this pass, also mark the next statement of each statement in the basic block. If there is no next statement, leave it as NULL. For for- and while-loops, the next statement of the last statement in the body is considered to be the test-part of the loop. This marking is made use of in the next step below.
3. By traversing the annotated parse tree, output the EDL case statement that gives behavior to the next state of the program counter, using function output_pc() for each statement. Function output_pc() works as follows:
   Inputs in addition to a pointer to the current statement are:
   – nextfather: statement following father statement
   – breaktowhere: statement to which to jump in case of a break statement
   – continuetowhere: statement to which to jump in case of a continue statement

   Note: The following makes use of the term "next statement" as described in the previous step.

   Note: In order to make the model concise, it is possible to suppress a line for groups of statements having similar behavior, and instead deal with them as a group. For instance, c2edl collects all return statements into a group and gives them behavior with a single guard (see the example in Section 7). It also suppresses a statement when the next value of the program counter is its own value plus one, and instead collects these into the default case as "pc+1". Also, if the model is of a multi-threaded program, the first line of the case statement needs to be a guard indicating whether or not this thread will progress this timestep. If not, the value of the program counter stays the same.

   Function output_pc() does the following: First output a guard which is the pc of the current statement. Then:
   – For an if statement:
     • Output an if-expression as follows: The condition is identical to the condition of the if statement (for simplicity here and in similar situations below we have ignored the case that the condition is an assigment or a list of statements). The then-part contains the pc of the then-part of the if statement. The else-part contains: if the if statement has an else-part, then the pc of the else-part of the if statement. Otherwise, if the if statement has a next statement, then the pc of the next statement. Otherwise, the pc of the next statement of the father statement.
     • Call output_pc() recursively for the then- and else-parts of the if statement. Parameter nextfather is the pc of the next statement, if it exists. Otherwise pass on nextfather as is. Parameters breaktowhere and continuewhere are passed on as is.

- For a case statement (consider each switch statement a separate statement): call output_pc() recursively for each of the switch statements. Parameter nextfather is the pc of the next statement, if it exists. Otherwise pass on nextfather as is. Parameters breaktowhere and continuewhere are passed on as is.
- For a switch statement:
  - Output the pc of the body of the switch.
  - Call output_pc() recursively for the body of the switch. Parameter nextfather is the pc of the next statement, if it exists. Otherwise pass on nextfather as is. Parameter breaktowhere is the current nextfather. Parameter continuewhere is passed on as is.
- For a for statement:
  - Call output_pc() recursively for the initialization-part of the if statement. Parameter nextfather is the pc of the current for statement. Parameters breaktowhere and continuewhere are 0.
  - Output an if-expression as follows: The condition is identical to the condition of the test-part of the for statement. The then-part contains the pc of the body of the for statement. The else-part contains: the pc of the next statement, if it exists, otherwise nextfather.
  - Call output_pc() recursively for the increment-part of the for statement. Parameter nextfather is the pc of the current for statement. Parameters breaktowhere and continuewhere are 0.
  - Call output_pc() recursively for the body of the for statement. Parameter nextfather is the pc of the increment-part of the for statement. Parameter breaktowhere is the pc of the next statement, if it exists. Otherwise it is the current nextfather. Parameter continuewhere is the pc of the increment-part of the current for statement.
- For a while statement:
  - Output an if-expression as follows: The condition is the condition of the while statement. The then-part is the pc of the body of the while statement. The else-part is the pc of the next statement of the while statement if it exists. Otherwise it is the pc of the current nextfather.
  - Call output_pc() recursively for the body of the while statement. Parameter nextfather is the pc of the while statement itself. Parameter breaktowhere is the pc of the next statement if it exists. Otherwise it is the pc of the current nextfather. Parameter continuewhere is the pc of the while statement itself.
- For a do statement:
  - Call output_pc() recursively for the body of the do statement. Parameter nextfather is the pc of the condition of the do statement. Parameter breaktowhere is the pc of the next statement if it exists. Otherwise it is the current nextfather. Parameter continuewhere is the pc of the condition of the do statement.
  - Output an if-expression as follows: The condition is the condition of the do statement. The then-part is the pc of the body of the do statement. The else-part is the pc of the next statement if it exists. Otherwise it is the pc of the current nextfather.

  - For a function call (not the push call), or an assignment statement, in increment or decrement statmeent, or an empty statement: if there is a next statement, then output its pc. Otherwise output the pc of the current nextfather. However, suppress in both cases if the value to be output is one more than the value of the current pc (this case is taken care of separately, see note regarding conciseness of code above).
  - For a return statement: do nothing. This case is taken care of separately. See note regarding conciseness of code above.
  - For a continue statement: output the value of continuewhere.
  - For a break statement: output the value of breaktowhere.
  - For a push call statement (dummy statement added by a previous step - see above): for simplicity, we assume that the next statement is a simple function call (as opposed to an assignment the righthand-side of which is a function call, or an if statement the condition of which uses a function call). Other cases can be dealt with by pre-processing, or directly. If the next statement is a simple function call: output the pc of the body of the function being called.

4. In the same manner, output an EDL case statement that gives the behavior as it would have been had there been no function calls. This is accomplished in the same way as above, but by ignoring the branching behavior of calls. The identifier *nextpcnocall* given behavior in this manner will be pushed on the stack.
5. Output the standardized behavior of the stacks and stack pointer as described above.
6. Output the value of *returntowhere*, which is a function of the stack and stack pointer. This is used to give a value to the program counter in the case that the current statement is a return statement.
7. Output behavior for *somereturn* and *somecall*, which indicate that the program counter has the value of some return statement or some push call statement, respectively.
8. Output behavior of the parameter array. This is an array containing value of the parameters of the current function, used by the parameter stack as described above.
9. Output behavior for each variable in the code, which is a case statement, the body of which is determined by once again traversing the parse tree, searching for assignments to the current variable. For each assignment statement found, output a guard with the corresponding program counter. The value assigned is determined by the right-hand-side of the assignment statement. If it is a function call, output *returnval*, described below. Otherwise, output an EDL expression corresponding to the C expression found. (The case where the C expression is not a single function call but rather an expression containing one or more function calls is dealt with by pre-processing, which breaks such assignments into multiple statements.)
10. Output the behavior of all values returned anywhere in the code by traversing the parse tree once again. The identifier *returnval* given behavior in

this manner is used by variables which are assigned values in assignment statements of the form $x = f(y)$.

7 A complete example

The source code of the software verified, consisting of approximately 2,500 lines of C, is confidential and thus and cannot be presented in full. In this section, instead, we show the complete translation process of a single function from the original source code (slightly modified to allow illustration of several points in one short example) to the manually modified pseudo-code, through the automatically generated EDL.

Figure 12 shows function *ca_dir_io_begin_start_read*(), and Figure 13 shows

```
static Ca_rc ca_dir_io_begin_start_read () {
    Ca_rc rc ;
    Ca_rc shared_rc ;
    Ca_rc buffer_rc ;

    rc = ca_dir_io_begin_hit_or_miss ( pio_wb ) ;
    if ( CA_RC_MISS == rc ) {
        pio_wb->dir_async_rc = CA_RC_MISS ;
        buffer_rc = ca_dir_io_begin_get_read_buffer ( pio_wb ) ;
        if ( CA_RC_WAIT == buffer_rc ) {
            rc = CA_RC_WAIT ;
        }
    } else {

        shared_rc = ca_dir_io_begin_get_shared ( pio_wb ) ;
        if ( CA_RC_WAIT == shared_rc ) {
            rc = CA_RC_WAIT ;
        }
    }
}
```

Fig. 12. Original code for function *ca_dir_io_begin_start_read*()

the corresponding pseudo-code, which was created manually. There is only one small difference between the original code and the pseudo-code - the modification of the pointer reference from $pio\_wb- > dir\_async\_rc$ to $dir\_async\_rc[pio\_wb]$ as discussed above. This illustrates the minimal amount of manual work that went into the translation process.

For the purposes of this example, the input to c2edl is the pseudo-code of function *ca_dir_io_begin_start_read*() shown in Figure 13. The output of c2edl is two files: the annotated (with *pc*) source code, and the EDL model. The annotated source code is shown in Figure 14 below[3]. The EDL model is not usually intended to be human readable. Therefore, the automatically generated EDL model shown in Figures 15 and 16 has been edited by adding

[3] The *pc* starts at 30 because only the annotation of function *ca_dir_io_begin_start_read*() is shown, without the annotations of the functions called by *ca_dir_io_begin_start_read*.

```
static Ca_rc ca_dir_io_begin_start_read () {
    Ca_rc rc ;
    Ca_rc shared_rc ;
    Ca_rc buffer_rc ;

    rc = ca_dir_io_begin_hit_or_miss ( pio_wb ) ;
    if ( CA_RC_MISS == rc ) {
        dir_async_rc[pio_wb] = CA_RC_MISS ;
        buffer_rc = ca_dir_io_begin_get_read_buffer ( pio_wb ) ;
        if ( CA_RC_WAIT == buffer_rc ) {
            rc = CA_RC_WAIT ;
        }
    } else {

        shared_rc = ca_dir_io_begin_get_shared ( pio_wb ) ;
        if ( CA_RC_WAIT == shared_rc ) {
            rc = CA_RC_WAIT ;
        }
    }
}
```

Fig. 13. Pseudo-code for function *ca_dir_io_begin_start_read*()

line breaks and indentation for the purposes of readability. In addition, for brevity, parts of the EDL model resulting from the body of functions called by *ca_dir_io_begin_start_read*() and not shown above have been edited out. Finally, again for brevity, repetitious parts of the code have been edited out. Anything that has been edited out is indicated by "...".

Line 1 declares the module *ca_dir_io_begin_start_read*() which will be instantiated by RuleBase (possibly along with other modules - recall that our program is multi-threaded). EDL puts input variables in the first set of parentheses and output variables in the second. Module *ca_dir_io_begin_start_read*() (and every other module created by c2edl) has a single input variable, *nogo*, and no output variables. The input variable *nogo* controls whether or not this thread will progress at the current time step (recall that our program is multi-threaded).

Lines 3 and 4 define *pcint* and *returntowhereint*, which are integer versions of the bit vectors *pc* and *returntowhere*. The EDL function *bvtoi*() converts a bit vector into an integer.

Lines 6 through 25 declare and give behavior to the bit vector *pc*. The size of the bit vector is automatically determined by c2edl as a function of the number of lines in the code. The initial value, assigned on line 7, is determined by the first line in the top-level function being translated. In our short example, it is the value corresponding to the first line in function *ca_dir_io_begin_start_read*() as per the annotated code of Figure 14. Line 8 assigns a value to the next state of the *pc* through a case statement. The first item in the case statement, on line 9, ensures that the *pc* does not change if the variable *nogo* has the value 1 (recall that *nogo* controls the interleaving in our multi-threaded model). Line 10 assigns the *pc* the value of *returntowhere* in the case that *somereturn*, indicating that the current command is a return, has the value 1 (*returnwhere* and *somereturn* are give values later, on lines 107 and 117, respectively.) Line 24 is the default, which is to increment the *pc*. Lines 11 through 23 give the value of the *pc* for

```
ca_dir_io_begin_start_read()
{
/* 30 */ /* 30 push call */;
/* 31 */ rc = ca_dir_io_begin_hit_or_miss(pio_wb);
/* 32 */ if (CA_RC_MISS == rc)
{
/* 34 */ dir_async_rc[pio_wb] = CA_RC_MISS;
/* 35 */ /* 35 push call */;
/* 36 */ buffer_rc = ca_dir_io_begin_get_read_buffer(pio_wb);
/* 37 */ if (CA_RC_WAIT == buffer_rc)
/* 39 */ rc = CA_RC_WAIT;
}
else
{
/* 40 */ /* 40 push call */;
/* 41 */ shared_rc = ca_dir_io_begin_get_shared(pio_wb);
/* 42 */ if (CA_RC_WAIT == shared_rc)
/* 44 */ rc = CA_RC_WAIT;
}
/* 45 */ return ;
}
```

Fig. 14. The annotated source code

branches and function calls in the body of the code. Only values of the *pc* within the range of function *ca_dir_io_begin_start_read*() are shown, others have been edited out for brevity. Thus we will start a detailed examination with line 12. Line 12 gives the next value of the *pc* when the *pc* currently has the value 30. This corresponds to the line marked 30 in the annotated code of Figure 14. Line 30 in Figure 14 shows a comment for a push call, which indicates that the next line is a function call. Thus, the next value of the *pc* is the value of the *pc* at the first line of the function being called, in this case function *ca_dir_io_begin_hit_or_miss*(), the first line of which has the value 0 (not shown). Line 13 gives the next value of the *pc* when the *pc* currently has the value 32. This corresponds to the line marked 32 in the annotated code of Figure 14. Line 32 in Figure 14 shows an if-statement, thus the next value of the *pc* in EDL is a corresponding if-expression (the local variable *rc* of function *ca_dir_io_begin_start_read*() has been renamed *rc_ca_dir_io_begin_start_read* by c2edl). The remainder of the code for the *pc* is similar.

Lines 27 and 28 define *maxpc* and *pcplusone*, used elsewhere.

Lines 30 through 42 give behavior to *nextpcnocall*, which is not a state variable, but rather a defined term, since its value is needed only at the current state (to put into the stack). It is similar in structure to lines 8 through 25, except that since we need it only to determine the value of the return location for the stack, it does not need entries corresponding to lines 9 and 10. The behavior corresponding to line 12, for instance, is covered by the default entry on line 41, because in the case that there had been no function call, the next value of the *pc* when the *pc* was 30 would have been 31, which is the return location for the call to function *ca_dir_io_begin_hit_or_miss*() on line 30 (the push call) of Figure 14 (at which point the assignment to *rc* will be executed).

Lines 44 through 46 declare the bit vector representing the stack pointer (*stackp*), a corresponding integer value, and the value of *maxstack*, which is an

```
module ca_dir_io_begin_start_read (nogo)() {

define pcint := bvtoi(pc(0..5));
define returntowhereint := bvtoi(returntowhere(0..5));

var pc(0..5): boolean;
assign init(pc(0..5)) := 30;
assign next(pc(0..5)) := case
   nogo: pc(0..5);
   somereturn: returntowhere(0..5);
   pc(0..5)=0: ...
   pc(0..5)=30:0;
   pc(0..5)=32:if ((CA_RC_MISS = rc_ca_dir_io_begin_start_read)) then 34 else 40 endif;
   pc(0..5)=35:22;
   pc(0..5)=37:if ((CA_RC_WAIT = buffer_rc_ca_dir_io_begin_start_read)) then 39 else 45 endif;
   pc(0..5)=39:45;
   pc(0..5)=40:5;
   pc(0..5)=42:if ((CA_RC_WAIT = shared_rc_ca_dir_io_begin_start_read)) then 44 else 45 endif;
   pc(0..5)=7: ...
   pc(0..5)=12: ...
   pc(0..5)=15: ...
   pc(0..5)=25: ...
   pc(0..5)=46:  ...
   else: if pcplusone>maxpc then itobv(maxpc) else itobv(pcplusone) endif;
esac;

define maxpc := 46;
define pcplusone := pcint+1;

define nextpcnocall(0..5) := case
   pc(0..5)=0: ...
   pc(0..5)=32:if ((CA_RC_MISS = rc_ca_dir_io_begin_start_read)) then 34 else 40 endif;
   pc(0..5)=37:if ((CA_RC_WAIT = buffer_rc_ca_dir_io_begin_start_read)) then 39 else 45 endif;
   pc(0..5)=39:45;
   pc(0..5)=42:if ((CA_RC_WAIT = shared_rc_ca_dir_io_begin_start_read)) then 44 else 45 endif;
   pc(0..5)=7: ...
   pc(0..5)=12: ...
   pc(0..5)=15: ...
   pc(0..5)=25: ...
   pc(0..5)=46:  ...
   else: if pcplusone>maxpc then itobv(maxpc) else itobv(pcplusone) endif;
esac;

var stackp(0..3): boolean;
define stackpint := bvtoi(stackp(0..3));
define maxstack := 5;

%for ii in 0..5 %do
var stack_%{ii}(0..5): boolean;
define stackint_%{ii} := bvtoi(stack_%{ii}(0..5));
var param_stack%{ii}(1.. 3): {param_types};
%end

assign init(stackp(0..3)) := 0;
       next(stackp(0..3)) := case
          nogo: stackp(0..3);
          somecall: if stackp(0..3)=6 then 6 else itobv(stackpplus1) endif;
          somereturn: if stackp(0..3)=0 then 0 else itobv(stackpminus1) endif;
          else: stackp(0..3);
       esac;

assign next(param_stack0(1)) := case
          nogo: param_stack0(1);
          somereturn & (stackp(0..3) = 1): {param_types};
          (0 != stackp(0..3)) | !somecall: param_stack0(1);
          else:  params(1);
esac;
...
assign next(param_stack1(1)) := case
          nogo: param_stack1(1);
          somereturn & (stackp(0..3) = 2): {param_types};
          (1 != stackp(0..3)) | !somecall: param_stack1(1);
          else:  params(1);
esac;
...
assign next(param_stack5(1)) := case
          nogo: param_stack5(1);
          somereturn & (stackp(0..3) = 6): {param_types};
          (5 != stackp(0..3)) | !somecall: param_stack5(1);
          else:  params(1);
esac;
...
```

Fig. 15. The EDL model - part 1

```

assign next(stack_0(0..5)) := case
        nogo: stack_0(0..5);
        somereturn & (stackp(0..3) = 1): nondets(6);
        (0 != stackp(0..3)) | !somecall: stack_0(0..5);
        else:  nextpcnocall(0..5);
     esac;
assign next(stack_1(0..5)) := case
        nogo: stack_1(0..5);
        somereturn & (stackp(0..3) = 2): nondets(6);
        (1 != stackp(0..3)) | !somecall: stack_1(0..5);
        else:  nextpcnocall(0..5);
     esac;
...
assign next(stack_5(0..5)) := case
        nogo: stack_5(0..5);
        somereturn & (stackp(0..3) = 6): nondets(6);
        (5 != stackp(0..3)) | !somecall: stack_5(0..5);
        else:  nextpcnocall(0..5);
     esac;

define stackpminus1 := if stackp(0..3) = 0 then 0 else bvtoi(stackp(0..3)) - 1 endif;
define stackpplus1 := if stackp(0..3) = 6 then 6 else bvtoi(stackp(0..3)) + 1 endif;

define returntowhere(0..5) := case
   stackpminus1=0:stack_0(0..5);
   stackpminus1=1:stack_1(0..5);
   stackpminus1=2:stack_2(0..5);
   stackpminus1=3:stack_3(0..5);
   stackpminus1=4:stack_4(0..5);
   stackpminus1=5:stack_5(0..5);
   else: 46;
esac;

define somereturn := (0|pc(0..5)=2|pc(0..5)=3|pc(0..5)=4|pc(0..5)=20|pc(0..5)=21|
                     pc(0..5)=27|pc(0..5)=28|pc(0..5)=29|pc(0..5)=45);

define somecall := (0|pc(0..5)=30|pc(0..5)=35|pc(0..5)=40);

var params(1..3): {param_types};
assign next(params(1)) := case
        nogo: params(1);
        somereturn & (stackpminus1 = 0): param_stack0(1);
        somereturn & (stackpminus1 = 1): param_stack1(1);
        somereturn & (stackpminus1 = 2): param_stack2(1);
        somereturn & (stackpminus1 = 3): param_stack3(1);
        somereturn & (stackpminus1 = 4): param_stack4(1);
        somereturn & (stackpminus1 = 5): param_stack5(1);
        pc(0..5)=30:pio_wb;
        pc(0..5)=35:pio_wb;
        pc(0..5)=40:pio_wb;
        else: params(1);
esac;
...

assign next(buffer_rc_ca_dir_io_begin_start_read) :=case
        nogo: buffer_rc_ca_dir_io_begin_start_read;
        pc(0..5)=36: returnval;
        pc(0..5)=45: {buffer_rc_ca_dir_io_begin_start_read_types};
        else: buffer_rc_ca_dir_io_begin_start_read;
esac;
...

assign next(returnval) := case
        nogo: returnval;
        pc(0..5)=2: CA_RC_HIT;
        pc(0..5)=3: CA_RC_MISS;
        pc(0..5)=20: rc_ca_dir_io_begin_get_shared;
        pc(0..5)=27: CA_RC_WAIT;
        pc(0..5)=28: CA_RC_SUCCESS_IMMEDIATE;
        else: {returnval_types};
esac;

}
```

Fig. 16. The EDL model - part 2

input to c2edl. For this example, the value of *maxstack* has been set to 5 (for the software examined in this paper, the value was set to 24, wihch was separately proved as sufficient to prevent a stack overflow).

Lines 48 through 52 declare:

1. The stack as a set of bit vectors (because EDL does not support two-dimensional arrays). The depth of the stack (seen on line 48) is a function of *maxstack*, hard-coded by c2edl into the resulting output file. The width of the stack is calculated by c2edl as a function of the maximum value of the program counter, and in this example happens to result in a stack as wide as it is deep.
2. Corresponding integer values of each stack entry
3. The parameter stack. The depth of the parameter stack is the same as that of the stack. The width of the vector is determined by the maximum number of parameters to any function in the code being translated. In our case the width 3 results from a call to a fuction not shown (it is called by one of the functions called by *ca_dir_io_begin_start_read*(). The vector *param_stack* is of an enumerated type, the values of which include all values which may be taken on by any parameter in the code. (Declaration of this enumerated type is manual, as are declarations of all other types used by the EDL model.)

Lines 54 through 60 give behavior to the stack pointer as described in Section 5.

Lines 62 through 67 give behavior to the first member (member 1) of the first parameter stack variable (*param_stack*0). This corresponds to the first parameter passed by a function, at stack depth 0. The code that was deleted for brevity at line 68 gives behavior to members 2 and 3. Similarly, lines 69 through 82 show behavior for the first member of the parameter stack variable at depths 1 and 5. Other parts of the parameter stack have been deleted for brevity.

Lines 84 through 89 give behavior to the stack at depth 0. Similarly, Lines 90 through 102 give behavior to depths 1 and 5, with the remaining parts of the stack deleted for brevity.

Lines 104 and 105 define *stackpminus*1 and *stackpplus*1, used elsewhere.

Lines 107 through 115 define *returntowhere*, used to select the depth of the stack at which the return location can be found.

Lines 117 through 118 define *somereturn*, used to control the behavior of the program counter and the stacks. For instance, the single return statement of function *ca_dir_io_begin_start_read*(), located at location 45, appears here. The other locations referred to appear in code called by *ca_dir_io_begin_start_read*() and not shown.

Line 120 defines *somecall*, used to control the stacks. In our case only the three push calls of function *ca_dir_io_begin_start_read*() are shown. Calls from functions called by function *ca_dir_io_begin_start_read*() do not appear because the body of the "grandchild" functions was not supplied to c2edl for the purpose of this example. In this case, c2edl considers the functions as noops and does not generate push calls for them.

Line 122 declares the state variables representing parameters, and lines 123 through 135 give behavior to the state variable representing the first parameter of the current function. If *nogo* has the value one, there is no change, as shown on line 124. If we are at a return statement, then the next value is taken from the parameter stack, where it was saved at the call. This is shown on lines 125 through 130. If we are at a function call, then the value is determined by the parameters passed in that call, as shown on lines 131 through 133. For instance, at line 30 in Figure 14, the first (and only) parameter passed in the call about to be made is *pio_wb*. This is shown on line 131. Finally, line 134 ensures that there is no change for any other type of statement. The code that was deleted for brevity at line 136 shows the behavior for other parameters.

Lines 138 through 143 give behavior to the local variable $buffer\_rc$ (renamed $buffer\_rc\_ca\_dir\_io\_begin\_start\_read$ by c2edl). On line 139, the value is held if nogo has the value one. Line 140 gives the value of *returnval* if the *pc* has the value 36, because on line 36 of Figure 14, $buffer\_rc$ gets the value returned by the currently returning function (*returnval* itself is given a value later, on line 146). Line 141 gives a non-deterministic value to $buffer\_rc$ in the case that the *pc* has the value 45, because at location 45 in Figure 14, the return statement renders the value of local variable $buffer\_rc$ undefined. Finally, line 142 ensures that $buffer\_rc$ changes its value nowhere else. The code that was deleted for brevity at line 144 gives behavior to other variables (local as well as global).

Finally, lines 146 through 154 give behavior to *returnval*, used by the code that gives behavior to the variables. If *nogo* has the value one, then *returnval* doesn't change (this is needed in case control switches to the other thread in between the time that *returnval* is assigned a value and the time it is used). Otherwise, the value of *returnval* is determined by the current return statement. If the current statement is not a return statement, *returnval* gets a non-deterministic value, as shown on line 153.

8 Special Reduction Problems of Software

One of the strengths of RuleBase is its ability to perform pre-model checking *reductions* on the design under verification. For instance, consider the function getminmax() shown in Figure 17 below. Suppose that the formula to be model checked is the following (where i is an auxiliary variable with the same range as variable a, and with the next state function $next\_i = i$):

$$AG((a > max \wedge a = i) \rightarrow AF(max = i)) \tag{3}$$

By examining Formula 3, and the next state functions of the variables appearing in it, RuleBase can determine that the variable *min* has no influence on the validity of the formula. Therefore, RuleBase can build the transition relation without using the next state function of variable *min*. This is known as a cone-of-influence reduction [7]. The cone-of-influence reduction is a simple reduction that is very powerful for hardware. However, in the world of software, it breaks down very quickly, as shown by the function lotsofbaggage() in Figure 18.

```
getminmax (){
  int min, max, a;

 a = max = 0; min = MAXINT;
 do {
   if (a > max)
     max = a;
   if (a < min)
     min = a;
   input(&a);
 } while(a);
 printf("min=%d,max=%d\n",min,max);
}
```

Fig. 17. minmax()

```
lotsofbaggage (){
  int min, max, a, b, x, y, z;

 b = x ? y : z;
 if (a > b)
   x = y;
 else
   x = z;
 a = max = 0; min = MAXINT;
 do {
   if (a > max)
     max = a;
   if (a < min)
     min = a;
   input(&a);
 } while(a);
 printf("min=%d,max=%d\n",min,max);
}
```

Fig. 18. lotsofbaggage()

Function lotsofbaggage() is identical to function getminmax(), except that before calculating the minimum and maximum, it does some assignments to variables b, x, and y, which are never used afterwards. Obviously, the values of variables b, x, and y do not affect the validity of Formula 3 for function lotsofbaggage(), and their next-state functions can be ignored. However, the cone-of-influence reduction used for hardware will not see this. The reason is that the values of a and of max, which are needed by the model checker, are a function of the pc, which is a function of b (because of line 1), which itself is a function of x and y (because of line 0).

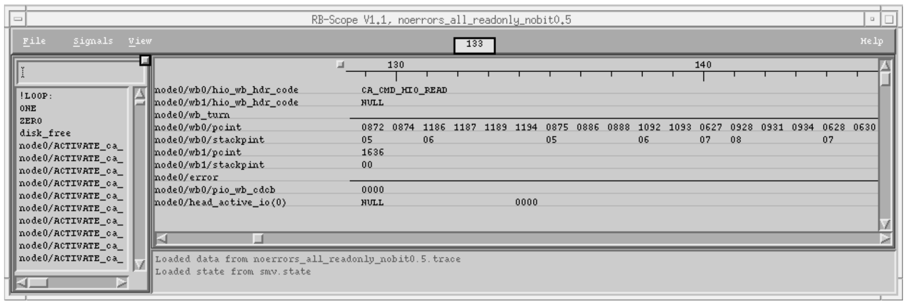
In the work described in this report, problems such as that illustrated by function lotsofbaggage() above were solved manually, by editing the code to eliminate the extraneous lines. This was a painful process, and probably many unnecessary variables were left in the model. Developing reductions specially suited to software, then, is an important direction for future work. An obvious place to start is with the extensive literature on program slicing [47, 16] and static analysis [41].

9 Results

The original goal of this work was to verify the correct behavior of the cache algorithms, including safety properties (data is cached correctly) as well as liveness properties (no deadlocks or livelocks resulting from resource management). That rather ambitious goal was set when it was still envisioned that an abstract model would be built manually. The list of properties changed during the course of the work, for the reason that precise modeling of resources (e.g. availability of a free buffer) turned out to be too expensive for the current state of software model checking. Therefore, resources were modeled as non-deterministicly being available or unavailable, and properties involving resources (e.g. no deadlock as a result of buffer allocation) were not checked.

Finally, due to size problems resulting from the decision to automatically create the model rather than hand-code it, even verifying correct functionality proved more difficult than originally expected. What was coded in the end was one node containing two work buffers (that is, allowing processing of two requests simultaneously). Other nodes were modeled non-deterministically. In retrospect, it might have been more fruitful to check two nodes with only one work buffer per node. The decision to code two work buffers on one node was made because experience with hardware designs has shown that there are usually more bugs resulting from the interaction of two requests on the same node than there are bugs resulting from the interaction of two requests on separate nodes.

The verification of the model resulted in a total of 12 traces showing possible cache algorithm problems. Traces contain a step-by-step path from the initial state of the system to a state which violates one of the formulas in the specification. To give a sense of the level of detail of these traces, a partial trace and a partial README file describing it appear in Figures 19 and 20 below. A trace is automatically created by RuleBase. Figure 19 is a screen capture of the window of tool Scope, which is used to display a trace. Scope shows the values of the variables as a function of time, which is indicated by the ruler at the top of the screen. Figure 19 shows a portion of the trace 013. The variables node0/wb0/pcint and node0/wb1/pcint indicate the program counter (as generated by the automatic translation, and available to the user in a file containing the annotated source code) of work buffers 0 and 1, respectively. Variables node0/wb0/stackpint and node0/wb1/stackpint indicate the depth of the call stack. Other variables, such as node0/wb0/hio_wb_hdr_code and node0/wb1/hio_wb_hdr_code, are those of the original program. Any variable can be displayed by clicking on it in the list of variables on the left hand side of the screen. The README file, a part of which is shown in Figure 20, is an interpretation of the trace intended as an aid to debugging, and is written manually.



```
!LOOP:
ONE
ZERO
disk_free
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_
node0/ACTIVATE_ca_

node0/wb0/hio_wb_hdr_code   CA_CMD_MIO_READ
node0/wb1/hio_wb_hdr_code   NULL
node0/wb_turn
node0/wb0/pcint             0872 0874 1186 1187 1189 1194 0875 0886 0888 1092 1093 0627 0928 0931 0934 0628 0630
node0/wb0/stackpint         05        06                  05             06        07   08                  07
node0/wb1/pcint             1636
node0/wb1/stackpint         00
node0/error
node0/wb0/pio_wb_cdcb       0000
node0/head_active_io(0)     NULL                0000

Loaded data from noerrors_all_readonly_nobit0.5.trace
Loaded state from smv.state
```

Fig. 19. Part of trace 013

```
at time 34, pc=1228, in start_read()
at time 39, pc=542, in read_dir_lookup()
at time 43, pc=1518, in dir_io_handler()
at time 45, pc=845, in dir_io_begin_track_access()
at time 113, pc=861, still in dir_io_begin_track_access(), just set cache_info pointer to 0
at time 130, pc=874, still in dir_io_begin_track_access(),
                        about to add wb0 to active_io queue of cache entry 0
at time 138, pc=1092, in dir_io_begin_process_cmd()
at time 140, pc=627, in dir_io_begin_start_read()
at time 145, pc=630, back from dir_io_begin_hit_or_miss(), got a miss
at time 158, pc=634, returning CA_RC_WAIT from dir_io_begin_start_read()
at time 166, pc=893, returning CA_RC_WAIT from dir_io_begin_track_access()
at time 169, pc=553, back from dir_io_handler(), waiting
at time 191, pc=1709, just reserved iob, about to call dir_resource_buffer_reserved()
at time 192, pc=375, in dir_resource_buffer_reserved()
at time 198, pc=361, in dir_io_buffer_allocated()
at time 200, pc=372, about to call dir_restart_enqueue()
at time 224, pc=1720, at top of restart queue, about to call read_dir_lookup_done()
                        dir_async_rc has the value CA_RC_MISS
```

Fig. 20. Partial README file of trace 013

Of the 12 traces generated as counter-examples by RuleBase, 9 were modeling errors, one (trace 005) was deemed a possible bug on a client node, one (trace 011) was a software bug showing a problem when the maximum number of sharers is reached, and one (trace 012) was a real problem in an n-way system.

With the exception of the three problematical traces (005, 011, 012), all rules passed for the configuration described above. Run time (on an IBM RS/6000 workstation model 270), memory usage, and model size is summarized in Table 21. There are two surprises in Table 21. First of all, the fact that the memory requirements of the compilation phase of RuleBase were 625M, while the memory requirements of the model checking itself were only 249M, is contrary to what is expected. Model checking is usually much more expensive in terms of memory requirements than any pre-processing phase. A reduction algorithm dedicated to software could probably do a better job at reduction while reducing the memory requirements considerably. The second surprise is that despite the long run time (28 CPU hours), the memory requirements of the model checking itself were quite small - there was no state space explosion. This was not true of early runs of the model, which exploded quickly. Additions of hand-coded hints [44] achieved this impressive result. The hints controlled the interleaving of the concurrent runs of the two work buffers, so that a fixed-point was first reached with control belonging only to the first work buffer, then only to the second, and so on. Only after a number of iterations were the hints released so that more complicated interleavings could be examined.

CPU time to translate the model from C to EDL	negligible
memory to translate the model from C to EDL	negligible
CPU time to compile the EDL model	3.5 minutes
memory use to compile the EDL model	625M
CPU time to perform the model checking	28 hours
memory to perform the model checking	249M
lines of pseudo-code in C model	2478
number of state variables in EDL model	362
number of reachable states	10^{150}

Fig. 21. Run time, memory usage, and model size

10 Future Work

It would have been nice if the current work had uncovered some more profound bugs in the cache algorithms. However, despite not achieving that goal, the verification work described in this report was useful, in that it showed that the ideas presented in [19] are practical for more than just the simple control flow problems examined in that work.

Now that this project has proved the viability of software model checking using RuleBase, it is clear that it is worthwhile to invest time in enhancements to RuleBase in order to better support software model checking. Alternatively, a dedicated software model checker could be built, which would use the same core model checking engines as RuleBase. Features to better support software model checking include:

1. Dedicated reduction algorithms for software as per Section 8.
2. Automated solution to the coding of complex data types such as structures, pointers, and unions.
3. Full support for local variables, in order to allow model checking of programs which require them to behave correctly for recursive calls.
4. A trace generation facility which more directly supports software. The graphical display of traces produced by RuleBase are oriented to hardware waveforms; reading off the value of the program counter from such a trace is a little unwieldy. For software, it would make more sense to present to the user a window similar to that of a graphical debugger such as xcdb, which would use the information generated by RuleBase to allow single-stepping forwards and backwards, as well as jumping over function calls, etc. A great portion of the README file could be generated automatically as well, providing automated aid to the interpretation of the trace.

11 Conclusion

This project has demonstrated the viability of software model checking for more than just simple control flow problems. While the level of manual intervention is the work described was relatively high, most of the manual steps could have been and will be in the future completely automated. What this project has demonstrated is that investing the effort in doing so is worthwhile. The level of detail displayed in the 12 traces provided to the software developers was greater than could have been expected judging by prevailing wisdom. It is usual to measure a formal verification project by the number of bugs it reveals, since proving correctness is usually impossible because of size problems. While the formal verification of the software cache algorithms uncovered only three minor problems, it was able to complete for the configurations run without state space explosion. This is an important result in itself. Finally, by developing specialized reduction algorithms dedicated to software, it seems clear that we will be able to significantly increase the size of designs that can be verified in this way.

The development of RuleBase was begun in 1994, when state-of-the-art model checking was a few tens of state variables. In the seven years since, the capacity of RuleBase has increased to the point where it is routinely used for hardware development and verification. Given the results of this project, it seems reasonable to expect that software model checking will be a mainstream tool for software development seven years hence.

Acknowledgements

Thank you to Jody Glider, for his willingness to invest the time of his team on support of this project, and to Larry Chiu and Paul Muench for providing that support. Larry's patience in introducing me to the software cache algorithms, and Paul's in examining numerous program traces in the months following, is greatly appreciated. Thank you to Sharon Barner and Ilan Beer for many fruitful discussions on the problems of modeling software. I also thank the anonymous reviewers whose suggestions greatly improved the quality of this paper.

References

1. Y. Abarbanel-Vinov, N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal, and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, 19(1):35–44, 2001.
2. R. Back. On correct refinement of programs. *Journal of Computer and Systems Sciences*, 23(1):49–68, August 1981.
3. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7^{th} International SPIN Workshop*, LNCS 1885. Springer-Verlag, 2000.
4. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM Gigahertz Processor: An abstraction algorithm for high-performance netlists. In *Proc. 11^{th} International Conference on Computer Aided Verification (CAV)*, LNCS 1633, pages 72–83. Springer-Verlag, 1999.
5. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13^{th} International Conference on Computer Aided Verification (CAV)*, LNCS 2102, pages 363–367. Springer-Verlag, 2001.
6. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase: Model checking at IBM. In *Proc. 9^{th} International Conference on Computer Aided Verification (CAV)*, LNCS 1254, pages 480–483. Springer-Verlag, 1997.
7. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *Proc. 33^{rd} Design Automation Conference (DAC)*, pages 655–660. Association for Computing Machinery, Inc., June 1996.
8. C. Bernardeschi, A. Fantechi, S. Gnesi, S. LaRosa, G. Mongardi, and D. Romano. A formal verification environment for railway signaling system design. *Formal Methods in System Design*, 12(2), March 1998.
9. A. Borälv and G. Stålmarck. Formal verification in railways. In M. Hinchey and J. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 329–350. Springer-Verlag, 1999.
10. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
11. E. Clarke and E. Emerson. Characterizing correctness properties of parallel programs as fixpoints. In *Seventh International Colloquium on Automata, Languages, and Programming*, LNCS 85. Springer-Verlag, 1981.

12. E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, , and L. Ness. Verfication of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proc. of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, April 1993.
13. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
14. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the* 22^{st} *International Conference on Software Engineering*, June 2000.
15. C. Demartini, R. Iosif, and R. Sisto. Modeling and validation of Java multithreading applications using SPIN. In *Proc.* 4^{th} *International SPIN Workshop*, 1998.
16. M. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proc. 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1999.
17. Á. Eiríksson. The formal design of 1M-gate ASICs. In *Second International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 1522, pages 49–63. Springer-Verlag, 1998.
18. C. Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. In *Proceedings 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 1703, pages 97–109, Bad Herrenalb, Germany, September 1999. Springer-Verlag.
19. C. Eisner. Model checking the garbage collection mechanism of SMV. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
20. C. Eisner. Using symbolic CTL model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):107–124, October 2002.
21. C. Eisner, R. Hoover, W. Nation, K. Nelson, I. Shitsevalov, and K. Valk. A methodology for formal design of hardware control with application to cache coherence protocols. In *Proc.* 37^{th} *Design Automation Conference (DAC)*, pages 724–729. Association for Computing Machinery, Inc., June 2000.
22. C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *Proceedings, 9th International SPIN Workshop on Model Checking of Software*, LNCS 2318. Springer-Verlag, 2002.
23. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc.* 12^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1855, pages 232–247. Springer-Verlag, 2000.
24. RuleBase User's Manual. Formal Methods Group, IBM Haifa Research Laboratory.
25. Guide to Sugar Formal Specification Language Version 1.3.1, November 2000. Formal Methods Group, IBM Haifa Research Laboratory.
26. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc.* 24^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Association for Computing Machinery, Inc., January 1997.
27. P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proc.* 9^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1254. Springer-Verlag, 1997.
28. A. Goel and W. Lee. Formal verification of an IBM Coreconnect Processor Local Bus arbiter core. In *Proc.* 37^{th} *Design Automation Conference (DAC)*, pages 196–200. Association for Computing Machinery, Inc., June 2000.

29. J. Groote, J. Koorn, and S. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. Logic Group Preprint Series 121, Utrecht University, 1994.
30. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.
31. G. J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc.* 7^{th} *International SPIN Workshop*, LNCS 1885, page 224 ff. Springer-Verlag, 2000.
32. G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Proc. PSTV/FORTE99*, pages 481–497. Kluwer, 1999.
33. Y. Kesten, A. Klein, A. Pnueli, and G. Raanan. Bridging the e-business gap through formal verification. In M. Hinchey and J. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 117–137. Springer-Verlag, 1999.
34. G. Leduc, O. Bonaventure, L. Léonard, E. Koerner, and C. Pecheur. Model-based verification of a security protocol for conditional access to services. *Formal Methods in System Design*, 14(2), March 1999.
35. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
36. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
37. J. Mertens. *Verifying the Safety Guaranteeing System at Railway Station Heerhugowaard.* PhD thesis, Utrecht University, 1996.
38. R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.
39. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.
40. J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
41. S. Muchnick and N. Jones. *Program Flow Analysis.* Prentice-Hall, 1981.
42. A. Parash. Formal verification of an MPEG decoder chip: A case study in the industrial use of formal methods. In *Proceedings of the Workshop on Advances in Verification (WAVe), (a post CAV-2000 workshop)*, Chicago, July 2000.
43. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. International symposium in Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.
44. K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *Proceedings 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 1703, Bad Herrenalb, Germany, September 1999. Springer-Verlag.
45. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In *Second International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 1522, pages 82–99. Springer-Verlag, 1998.
46. S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc.* 7^{th} *International SPIN Workshop*, LNCS 1885, page 224 ff. Springer-Verlag, 2000.
47. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
48. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the* 15^{th} *International Conference on Automated Software Engineering*, Grenoble, France, September 2000.
49. K. Yorav, S. Katz, and R. Kiper. Reproducing synchronization bugs with model checking. In T. Margaria and T. Melham, editors, *Proceedings 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2144. Springer-Verlag, 2001.