Electronic Notes in Theoretical Computer Science 55 No. 3 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume55.html pages

Model Checking the Garbage Collection Mechanism of SMV

Cindy Eisner [1]

IBM Haifa Research Laboratory
Matam Advanced Technology Center
Haifa, 31905 Israel

Abstract

This paper describes an experience in the application of the RuleBase model checker to software written in C, using the tool c2edl. C2edl translates ANSI-C code to EDL, the input language of RuleBase. Although c2edl uses a radical abstraction in order to address the problems of software model checking, the abstract model built by c2edl proved sufficient to allow analysis of the garbage collection mechanism of SMV. Using c2edl and RuleBase, eight bugs were found in RuleBase itself, which uses the same garbage collection mechanism.

1 Introduction

In recent years, 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 [16,3,24,19,34,1,38]. In the past few years, there has been increasing interest in the application of model checking to software. One approach [22,23,36,20,2,37] is to develop new techniques which are specialized for software. A second approach [15,26,28,27,14] is to use modeling techniques that allow the application of existing tools. The advantage of the former is that it allows the difficulties inherent in software model checking to be addressed directly, while the advantage of the latter is that years of development and optimization effort put into an existing tool do not go to waste. In this paper, we describe an experience with the second approach, using the tool c2edl. C2edl translates ANSI-C code to EDL, the input language of the RuleBase model checker [6]. C2edl uses a radical abstraction in order to address the problems of software model checking. Nevertheless, the abstract

[1] Email: eisner@il.ibm.com

model built by c2edl proved sufficient to allow analysis of the garbage collection mechanism of SMV. Using c2edl and RuleBase, eight bugs were found in RuleBase itself, which uses the same garbage collection mechanism.

The remainder of this paper is organized as follows. Section 2 compares this work with related work. Section 3 gives some background on RuleBase. Section 4 describes how a program can be represented in a form suitable for symbolic model checking. Section 5 presents the tool c2edl. Section 6 discusses the application of model checking to the garbage collection mechanism of SMV, and gives experimental results. Section 7 concludes and points to future directions for research.

2 Comparison with related work

Many previous works have described the process of verifying high level models of software [12,29,30]. In this paper, we apply model checking to the source code itself, by means of an automatically generated abstraction, 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 [25,33,10,11,18,17]. 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 [35]. In this paper, we apply model checking to software written in the general purpose language C.

Godefroid [22,23] 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 [22,23] is the search algorithm, which performs a variety of explicit state space exploration. Stoller [36] takes an approach similar to that of [22,23] 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 [26] 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 [26], 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 [15,26] 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.

Visser, Havelund, Brat and Park present the second generation of Java PathFinder in [37]. 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 used modeling techniques to allow the application of an existing one.

Holzmann and Smith [28] present a method for extracting verification models from source code that results, as here, in a control-flow skeleton. However, their abstraction process is only semi-automatic, and is aided by a lookup table and model template manually coded by the user. In contrast, the use of c2edl does not require manual intervention. 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 [27], Holzmann describes another application of the method to a checkpoint management system. Again, the size of the software is not discussed.

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, c2edl is completely automatic, and we present results for a non-trivial application.

Esparza, Hansel, Rossmanith and Schwoon [20] describe model checking algorithms for pushdown automata. They take, as we do, the radical approach of abstracting away all variable values. However, they are not limited to a finite stack. In contrast, c2edl produces a finite state model for RuleBase. They give impressive results for randomly generated flow graphs (skeleton programs) of up to 20,000 lines.

Finally, Ball and Rajamani [2] 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. In contrast, we check properties expressed in temporal logic. Their approach to the semantic difficulties of software is to limit all variables to boolean values, and procedure calls to call-by-value parameters. Like [20], they are not limited to finite state systems. They show results for a simple family of programs with increasingly deep levels of nested procedure calls, but limited non-determinism. In contrast, we put an artificial limit on the level of nesting, but easily deal with a high level of non-deterministic behavior.

3 Preliminaries

The work described in this paper was performed using RuleBase [6]. RuleBase was originally based on a version of SMV [32]. After eight years of development [4,8,3,9,7,5,6,21], the original SMV code is a small part of the whole. Nevertheless, the garbage collection mechanism of SMV remains.

The input language of RuleBase is EDL, a dialect of the language SMV. RuleBase uses the temporal logic Sugar [4] as its specification language. Sugar combines the power of regular expressions with a syntactic sugaring of CTL. The work described in this paper uses only one kind of Sugar formula, a suffix implication. Informally, such a formula consists of two parts: a *sequence* and a *required condition*. A sequence is a finite prefix of a computation path, described as a regular expression. A required condition is a Sugar formula which is required to hold in every final state of the sequence. For example, the following Sugar formula:

$$\{\mathbf{true}[*], a, b[*], c[+]\}(d) \tag{1}$$

states that d is required to hold at the final state of every sequence described by the regular expression $ab*c+$. The equivalent CTL formula is:

$$\neg EF(a \wedge EX\ E[b\ U\ E[c\ U\ (c \wedge \neg d)]]) \tag{2}$$

4 Expressing a program as a set of next-state functions

The process of translating a program to a set of next-state functions suitable for model checking is similar to that of [31,13]. We describe it informally for a simple example, then add details. Consider the C function getmax() of Figure 1. We annotate the code with the value of the program counter (pc). If we restrict the integers a and max to the range 0 through 3, we can then

```
   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. 1. Function getmax() in C

rewrite getmax() in terms of next-state functions of the variables, as shown in Figure 2. We have expressed the call to $a = input()$ as a non-deterministic assignment to the variable a. The next state functions of the other variables are deterministic. For simplicity, we have ignored the details of translating the return statement (an example including function calls appears below). With minor syntactic changes and the addition of state variable declarations, Figure 2 is a complete SMV or EDL program, and can be model checked using SMV or RuleBase.

```
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. 2. Function getmax() in terms of next-state functions

```
 doit() {
   int max;
8   max = getmax();
9 }
```

Fig. 3. Program doit()

Of course, an interesting C program will typically be more complicated than function getmax(). Extending the translation to other kinds of branching and loop statements is straightforward. However, the translation process should also be able to deal with complex data types, pointers, and function calls, including recursive function calls. All of these could be dealt with by mimicking a compiler, or by starting the translation from assembly or machine code [2]. However, such a solution would be purely theoretical, since the state explosion problem would make it impossible to model check all but the most trivial programs. In the next section, we describe the solution used by c2edl.

5 C2edl

The solution to the semantic problems of modeling software used by c2edl is a radical abstraction which is easily automated. C2edl eliminates all variables except for the program counter and a finite stack, and replaces references to the variables with non-deterministic choice (i.e., *if* $(a > max)$ becomes *if* $\{0, 1\}$). The result is a skeleton program that represents an over-approximation of all possible control flows of the original program. For instance, consider program doit() shown in Figure 3 which calls function getmax() of Figure 1. Its abstraction is shown in Figure 4. Since all variables are eliminated, complex data types and pointers do not require special treatment. There is no need to save the values of local variables on the stack (because there are none). Thus, to support function calls, including recursive calls, it is enough to save the program counter. The stack is limited to a finite (and small) depth by use of

[2] Recursion would still be problematical in theory, since it is potentially infinite. We can ignore this problem if we assume that our software is running on some real machine, with a finite stack.

```
next(pc) = if pc=0 then 1
           else if pc=1 then 2
           else if pc=2 then if {0,1} then 3 else 4
           else if pc=3 then 4
           else if pc=4 then 5
           else if pc=5 then if {0,1} then 1 else 6
           else if pc=6 then stack(stackp-1)
           else if pc=7 then 7
           else if pc=8 then 0
           else if pc=9 then 9
next(stackp) = if pc=8 then stackp_inc
               else if pc=6 then stackp_dec
               else stackp
next(stack(stackp)) = if pc=8 then 9
                      else stack(stackp)
stackp_inc = if stackp = max_stackp then stackp else stackp+1
stackp_dec = if stackp = 0 then stackp else stackp-1

invar stackp <= max_stackp
```

Fig. 4. Program doit() abstracted

```
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)
```

Fig. 5. Standardized behavior of the stack

an invariant.

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. During this traversal, 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. These are used to standardize the behavior of the stack as shown in Figure 5. In addition to propositions *somecall*, *somereturn*, and *nextpcnocall*, c2edl automatically generates propositions of the form $assign\_v$ (indicating an assignment to variable v), use_v (indicating a use of variable v) and $call\_f$ (indicating a call to function f) for each variable v and function f in the program. This can be done without adding additional variables, because each of these propositions can be expressed purely as a function of the program counter. The complete output of c2edl for program doit() is shown in Appendix A.

At first glance, it seems that the abstraction described rids the model of all meaning. Indeed, the interesting properties of getmax() can not be verified using the abstracted model shown in Figure 4. However, there are programs for which the abstraction preserves enough information to be useful. The garbage collection mechanism of SMV is one such example. The process of model checking it is discussed in the next section.

6 Model Checking SMV

Using a model built by c2edl, the usage of the garbage collection mechanism was checked for SMV version r2.4.4, and for RuleBase, which uses the same mechanism. C2edl was invoked in a mode in which bit vectors are used instead of integers (see Appendix A), with the stack limited to a depth of 5.

6.1 The Garbage Collection Mechanism of SMV

The model checker SMV uses binary decision diagrams (BDDs) as its basic data structure. Since memory usage is a problem, it is necessary to periodically discard BDDs which are no longer needed. This is the job of the garbage collection mechanism. It works as follows. Garbage collection is performed at various places in the code by explicit calls to the function mygarbage(). During garbage collection, every BDD not marked to be saved is collected and discarded. A BDD is saved by a call to the function save_bdd(), which puts the saved BDD on a linked list called the save_bdd_list, and returns its argument. For instance, BDD v is safe from collection after a call to save_bdd(v), and after the call v=save_bdd(u), where u is some other BDD. A BDD is removed from the list by a call to release_bdd(). For instance, BDD v is released by the call release_bdd(v). There may be several occurrences of the same BDD on the save_bdd_list. Function save_bdd() always adds one occurrence, and release_bdd() always deletes one.

If a BDD which is needed for a future computation is collected as garbage, the result is a *dangling reference*, i.e., a BDD which potentially contains junk. If a BDD which is not needed for future computation is never released, the result is a *memory leak*, a needless blowup in memory requirements.

The problem of a dangling reference can be stated thus for BDD v: if a value is assigned to v without a call to save_bdd(), and if it is not saved by a call to save_bdd() before the next call to mygarbage(), and if it is then used by another calculation, this is an error. Since we have the following atomic propositions (generated automatically as described in Section 5):

- assign_v: an assignment to v
- call_save_bdd_v: a call to save_bdd() with v as an argument, or a call to save_bdd() the result of which is assigned to v
- call_mygarbage: a call to mygarbage()
- use_v: a use of v

we can express the requirement that there are no dangling references in Sugar as follows:

$$\{\mathbf{true}[*], assign\_v \wedge \neg call\_save\_bdd\_v, \neg(assign\_v \vee call\_save\_bdd\_v)[*], call\_mygarbage, \neg assign\_v[*], use\_v\}(\mathbf{false}) \tag{3}$$

The problem of a memory leak for BDD v can be stated as follows: if BDD v is saved by a call to save_bdd(), and then another value is assigned to

v without an intervening call to release_bdd(), this is an error. With the aid of the atomic propositions above, and the additional proposition following:

- call_release_bdd_v: a call to release_bdd() with v as an argument

we can express this in Sugar as follows:

(4) $\{\mathbf{true}[*], call\_save\_bdd\_v, \neg call\_release\_bdd\_v[*], assign\_v\}(\mathbf{false})$

6.2 Experimental results

C2edl and Formulas 3 and 4 were used to model check the garbage collection mechanism of SMV. The function build_symbols() of file symbols.c was checked for version r2.4.4 of SMV. Any function calls to functions also appearing in symbols.c were translated, other function calls were ignored. The generated model consisted of 3953 lines of code (that is, the value of the program counter ranged from 0 to 3952). There are 178 variables of type BDD in symbols.c. For each, Formulas 3 and 4 were generated, for a total of 356 formulas. In RuleBase, formulas can be grouped into *rules*. The formulas were checked on-the-fly [9] in 16 groups of 22-24 formulas per rule. Table 1 shows results for rules build_symbols0 through build_symbols15. All fails shown are false negatives as described in the next section.

RuleBase itself was checked as well. In particular, a function called reduction(), which was being debugged at the time, was checked. The generated model consisted of 2630 lines of code (that is, the value of the program counter ranged from 0 to 2629). Out of 352 formulas checked, 47 failed. Of those, 39 were false negatives as described in the next section, and 8 were real problems with the use of the garbage collection mechanism. The used of c2edl allowed these problems to be found statically using RuleBase, before the usual regression testing of a new version had begun. While problems with the use of the garbage collection mechanism are usually very painful to debug, the use of c2edl and RuleBase allowed them to be found and fixed easily. Instead of an unexpected result or a mysterious segmentation violation, which is the indication of a test gone wrong, the counter-examples generated pointed precisely to the source line (as indicated by the program counter) exhibiting the problem.

6.3 False positives and false negatives

The utility of Formulas 3 and 4 is highly dependent on the coding style used by the programmer. For instance, the code fragment of Figure 6 is safe in that

```
...
b = save_bdd(a);
c = b;
mygarbage();
d = c;
...
```

Fig. 6.

the value of c is not corrupted by the call to mygarbage(). However, it will be

Table 1
Results for SMV version 2.4.4

rule name	vars	run time	memory	# formulas in rule	# failing formulas
build_symbols0	91	19250 s	141 MB	22	0
build_symbols1	91	68580 s	303 MB	22	2
build_symbols2	91	28482 s	193 MB	22	1
build_symbols3	91	82946 s	321 MB	22	2
build_symbols4	91	9048 s	123 MB	22	0
build_symbols5	91	13692 s	154 MB	22	1
build_symbols6	91	15432 s	149 MB	22	2
build_symbols7	91	27076 s	203 MB	22	3
build_symbols8	91	42793 s	249 MB	22	3
build_symbols9	91	33952 s	216 MB	22	3
build_symbols10	91	11186 s	130 MB	22	2
build_symbols11	91	18019 s	168 MB	22	3
build_symbols12	91	43809 s	247 MB	22	3
build_symbols13	91	123990 s	424 MB	22	5
build_symbols14	91	99560 s	366 MB	24	7
build_symbols15	91	42704 s	276 MB	24	4

flagged as a violation by Formula 3 because Formula 3 does not "know" that variable c was assigned a value previously saved as b.

Another problem is illustrated by Formula 4. It will erroneously flag violations for local variables the use of which is in fact safe. For instance, consider function f() and the code fragment which calls it in Figure 7. The call to

```
bdd_ptr f(bdd_ptr p,q) {
0   p = save_bdd(g(p,q));
1   return(p);
2 }

...
10  a = f(b,c);
11  mygarbage();
12  release_bdd(a);
13  d = g(a);
...
```

Fig. 7.

release_bdd() on line 12 releases the value which was saved inside of function f() on line 0. However, Formula 4 will flag a second call to function f() as a violation, because it does not "see" a call to release_bdd() for signal p (the call on line 12 releases signal a) between the two assignments to variable p from two separate calls to the function. These and other false negatives can be avoided by adherence to certain coding conventions.

We now turn to the problem of false positives. Firstly, Formulas 3 and 4 probably do not completely express the correct use of the garbage collection mechanism of SMV. The second problem is more serious. Since we have limited the depth of our stack, we will not find errors which occur for only deeper levels of nested calls. This is a fundamental problem of the model of software as we have described it. The problem of false positives means that our method cannot be used for the verification of software. However, false positives are not a barrier to the use of the technique in the falsification of software. It is in the practical light of falsification, then, that this work should be viewed.

7 Conclusions and future work

We have described an experience in the application of symbolic model checking to general purpose software, the garbage collection mechanism of the model checker itself. In the process, eight real bugs were found in a version of the model checker under development. While the method is not suitable for verification, because of the potential for false positives, the experimental results show that it is extremely useful in the process of falsification.

Future work includes applying the method or variations on it to other applications. In addition, it would be interesting to experiment with infinite state techniques such as those described in [20], which do not require limiting the depth of the stack.

Acknowledgements

My original model of the garbage collection mechanism of SMV was much more complicated than that described in this paper. Thank you to Ilan Beer, who pointed out that the mechanism and its correct use could be expressed solely as a function of the program counter. Thank you to Shoham Ben-David, Avigail Orni, and Yaron Wolfsthal for their time reviewing and for helpful comments.

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), 2001. to appear.

[2] 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.

[3] 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.

[4] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc.* 13^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS. Springer-Verlag, 2001. to appear.

[5] 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. Springer-Verlag, 1997.

[6] 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.

[7] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc.* 9^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1254, pages 279–290. Springer-Verlag, 1997.

[8] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2), 2001.

[9] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc.* 10^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.

[10] 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.

[11] 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.

[12] 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.

[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] 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.

[17] 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).* to appear.

[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, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[19] 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.

[20] 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.

[21] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proc.* 6^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 818, pages 299–310. Springer-Verlag, 1994.

[22] 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.

[23] 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.

[24] 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.

[25] J. Groote, J. Koorn, and S. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. Logic Group Preprint Series 121, Utrecht University, 1994.

[26] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[32] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[33] J. Mertens. *Verifying the Safety Guaranteeing System at Railway Station Heerhugowaard.* PhD thesis, Utrecht University, 1996.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] K. Yorav, S. Katz, and R. Kiper. Reproducing synchronization bugs with model checking. In *Proceedings 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS. Springer-Verlag, 2001. to appear.

A Output of c2edl for program doit() of Figure 3

Below is the full output of c2edl for program doit() of Figure 3. C2edl outputs two files, *.cout and *.edl. The *.cout file is the original C code annotated with the program counter as allocated by c2edl. The *.edl file is the RuleBase model of the C code, in the language EDL, the input language of RuleBase.

Figure A.1 shows the file doit.cout. Note that the program counter as allocated by c2edl differs the numbering used in Figures 1, 3 and 4. Figure A.2 shows the file doit.edl, the model built by c2edl from program doit(). The model shown in Figure A.2 uses integers for the program counter and stack; c2edl has a switch which allows these to be modeled using bit vectors instead. The depth of the stack is controlled by a parameter. In the example shown here, the depth was set to 5.

```
getmax()
{
/* 0 */ a = /* 0 */ max = 0;
/* 1 */ do {
/* 2 */   if (a > max)
/* 4 */     max = a;
/* 5 */   /* push call */;
/* 6 */   a = input();
/* 7 */ } while (a);
/* 8 */ return max;
/* 9 */ return ;
}
doit()
{
/* 10 */ /* push call */;
/* 11 */ max = getmax();
/* 12 */ return ;

}
```

Fig. A.1. File doit.cout

```
var pc: 0..13;
assign init(pc) := 10;
assign next(pc) := case
somereturn: returntowhere;
pc=2:if pcaux=0 then 4 else 5 endif;
pc=7:if pcaux=0 then 2 else 8 endif;
pc=10:0;
pc=13: 13;
else: if pcplusone > maxpc then maxpc else pcplusone endif;
esac;
define maxpc := 13;
define pcplusone := pc+1;
var pcaux: 0..2;
define nextpcnocall := case
pc=2:if pcaux=0 then 4 else 5 endif;
pc=7:if pcaux=0 then 2 else 8 endif;
pc=13: 13;
else: if pcplusone > maxpc then maxpc else pcplusone endif;
esac;
var stackp: 0..6;
%for ii in 0..5 %do
var stack_%{ii}: 0..13;
assign init(stackp) := 0;
      next(stackp) := case
                      somerealpushcall: if stackp=6 then 6 else stackp + 1 endif;
                      somereturn: if stackp=0 then 0 else stackp - 1 endif;
                      else: stackp;
      esac;
invar stackp != 6;
var auxnondet: 0..13;
assign next(stack_0) := case
                        somereturn & (stackp = 1): auxnondet;
                        (0 != stackp) | !somerealpushcall: stack_0;
                        else: nextpcnocall;
      esac;
assign next(stack_1) := case
                        somereturn & (stackp = 2): auxnondet;
                        (1 != stackp) | !somerealpushcall: stack_1;
                        else: nextpcnocall;
      esac;
assign next(stack_2) := case
                        somereturn & (stackp = 3): auxnondet;
                        (2 != stackp) | !somerealpushcall: stack_2;
                        else: nextpcnocall;
      esac;
assign next(stack_3) := case
                        somereturn & (stackp = 4): auxnondet;
                        (3 != stackp) | !somerealpushcall: stack_3;
                        else: nextpcnocall;
      esac;
assign next(stack_4) := case
                        somereturn & (stackp = 5): auxnondet;
                        (4 != stackp) | !somerealpushcall: stack_4;
                        else: nextpcnocall;
      esac;
assign next(stack_5) := case
                        somereturn & (stackp = 6): auxnondet;
                        (5 != stackp) | !somerealpushcall: stack_5;
                        else: nextpcnocall;
      esac;
define stackpminus1 := if stackp = 0 then 0 else stackp - 1 endif;
define stackpplus1 := if stackp = 6 then 6 else stackp + 1 endif;
define 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;
else: 13;
esac;
define useof_a_getmax := (0|(pc=2)|(pc=4)|(pc=1));
define useof_max_doit := (0);
define useof_max_getmax := (0|(pc=2)|(pc=8)|(pc=8));
define useastopparam_a_getmax := (0);
define useastopparam_max_doit := (0);
define useastopparam_max_getmax := (0);
define useastopparam_ := (0);
define assignto_a_getmax := (0|(pc=0)|(pc=6));
define assignto_max_doit := (0|(pc=11));
define assignto_max_getmax := (0|(pc=0)|(pc=4));
define assignto_ := (0);
define someassign := (0|assignto_a_getmax|assignto_max_doit|assignto_max_getmax);
define callto_getmax := (0|(pc=11));
define callto_input := (0|(pc=6));
define callto_doit := (0);
define somerealcall := (0|callto_getmax|callto_doit);
define somereturn := (0|pc=8|pc=9|pc=9|pc=12|pc=12);

define somerealpushcall := (0|pc=10);
```

Fig. A.2. File doit.edl