Using a Hardware Model Checker to Verify Software

Stephen A. Edwards*
CS Department, Columbia University
500 West 120th Street
New York, New York, 10027, USA
sedwards@cs.columbia.edu

Tony Ma
Synopsys Advanced Technology Group
700 E Middlefield Rd
Mountain View, California, 94043, USA
tonyma@synopsys.com

Robert Damiano
Synopsys Advanced Technology Group
19500 NW Gibbs Dr #300
Beaverton, Oregon, 97006, USA
robertd@synopsys.com

Abstract

A variety of new algorithms has begun to enable model checking of industrial-sized netlists. This work attempts to apply that technology to the verification of embedded software: C programs that manipulate integers and contain unstructured control flow, but are not recursive and do not dynamically allocate memory.

We describe a synthesis procedure for translating a subset of C into a netlist and present experiments that show the models it builds seem to be harder to verify than typical hardware circuits, suggesting the problem has a different character.

Although we only have preliminary experimental results, they help to identify the challenges inherent in verifying this class of software and leave open the possibility of more successful approaches.

1 Introduction

Finite-state model checking of hardware—logic netlists—has grown powerful by employing a variety of algorithms that include symbolic state-space traversal, bounded model checking (e.g., satisfiability-based algorithms), and the clever combination of these algorithms. Such a combination has allowed the Ketchum model checker [8, 14] to prove properties of and generate test cases for industrial-sized circuits that were previously beyond reach. This work attempts to answer whether this technology can also be applied to checking embedded software.

Embedded software is a vague term, but here we take it to mean finite-state software containing a blend of arithmetic and decision making. Most embedded software is written in C, so we attempt to verify programs written in a subset that includes integer arithmetic, unstructured control-flow, and array accesses, but prohibits dynamic memory allocation and recursion. Ultimately, we also intend to address embedded software's penchant for I/O, but for the moment we are focusing on the behavior during the many cycles between I/O operations.

This paper makes two contributions: it describes a synthesis procedure that builds a hardware model (a netlist) for C programs written in a subset of the language, and it presents experiments that suggest the netlists this procedure builds are harder to verify than the hardware circuits for which Ketchum was designed. From this, we conclude that further progress will come only after changing the software we are attempting to verify, the synthesis procedure, or the Ketchum model checker's algorithms.

The Ketchum model checker [8, 14] takes a netlist and attempts to generate tests for the reachable combinations of user-supplied coverage signals and prove the others are unreachable. It employs a variety of techniques including random simulation, satisfiability (SAT), automatic test pattern generation (ATPG), symbolic state space traversal, and symbolic simulation. To verify software—a C function—we build a netlist that models its behavior and ask Ketchum to check signals corresponding to *assert* statements. Ketchum may generate a test to show an *assert* can run and fail, which helps the programmer diagnose the error; generate a test that makes the *assert* run and pass, which may be suitable for a regression suite; or prove that the *assert* can never run and fail, increasing a programmer's confidence in program correctness.

The synthesis procedure that builds the netlist from a C function for Ketchum resembles behavioral synthesis algorithms (see, e.g., De Micheli [7]), but its objectives differ. Unlike synthesis for a hardware implementation, which focuses on meeting worst-case cycle time, this procedure minimizes the number of intermediate states by chaining as many operations in a clock cycle as possible.

Currently the synthesis procedure only handles a subset of C, but more of the language could be accommodated. Integer arithmetic, unstructured control-flow (i.e., all the standard control-flow constructs such as if-then, for, and while are supported as well as gotos), and arrays are currently supported: a large enough subset to be represen-

*The author did this work while at Synopsys.

tative of much embedded software. Each array is modeled by a separate memory object (i.e., with read and write ports, each with an address). Pointers could be supported using the techniques of Séméria et al. [11], who perform pointer analysis to determine to which memory areas each pointer may refer to transform each pointer dereference into an array access.

Since Ketchum only handles finite-state models, we do not support recursion or dynamic memory allocation. These are not significant restrictions as most embedded software does not use recursive algorithms for fear of running out of stack space, nor does it do extensive memory allocation for fear of running out of memory.

1.1 Related Work

The synthesis procedure combines ideas from existing approaches. We synthesize the datapath and its multiplexers using the well-known static single assignment form [6]. Controller synthesis uses a simplified form of Berry's technique for Esterel [4], which also handles preemption and parallel synchronization. Sharp and Mycroft's [12] on-demand technique to insert clock cycles inspired ours, and they use a similar technique to synthesize controllers.

The Bandera project from the University of Kansas [5] takes a similar approach, also transforming an unmodified software program (in their case, a concurrent Java program) into a form suitable for existing model checkers. Instead of the netlist form used here, they use a guarded command language as an intermediate representation. They also perform automatic (e.g., slicing) and manual abstraction based on the property being tested.

Many techniques have been developed specifically for concurrent software. Holtzmann's SPIN system [9] is one of the more successful. Because it represents its reached states explicitly, it is able to handle more dynamic behavior such as process creation. For speed, it compiles code that evaluates the next state function. By contrast, Ketchum represents its states implicitly and evaluates the next state by simulating an in-memory netlist.

Ball and Rajamani [2, 3] take a hybrid approach where they represent a program's control-flow explicitly but the states at each control point implicitly. They use an interprocedural dataflow analysis algorithm from the compiler community to handle recursion.

2 The Synthesis Procedure

Figure 1 shows how the synthesis procedure builds a hardware model for the greatest common divisor function in Figure 1a. The procedure builds a control-flow graph (CFG), marks certain control arcs as clock cycle boundaries, determines which variables to save and restore across clock cycles, and casts the program into static single-assignment form. Finally, the controller is generated from the CFG and the structure of the datapath is read from the static single-assignment form.

2.1 Generating the Control-Flow Graph

The first step transforms the function being modeled into a CFG whose nodes contain sequences of assembly-language-like three-address instructions (Figure 1b). These typically consist of an operator such as addition, two source operands, and a destination, much like a typical datapath element. This step is essentially a standard compiler front end, which we implemented using the SUIF 2 system developed by Lam and her students at Stanford with the MachSUIF extensions [13].

The opcodes are mnemonics. In n_2, "sne r_0, a, b" ("set not equal") performs the `a != b` comparison from line 3, setting r_0 to 1 if a and b are different, and "bt n_3, r_0" means to branch to n_3 if r_0 is non-zero. Similarly, sgt in n_3 (set greater than) computes `a > b` from line 4. The two arithmetic instructions in node n_4 and n_5 perform arithmetic, e.g., in n_5, "sub b, b, a" computes `b -= a` in line 7.

The next two steps address the two problems that prevent hardware from being generated from the CFG: loops and multiple writes to the same register.

2.2 Identifying Clock Cycle Boundaries

An arithmetic circuit such as an adder cannot perform more than one operation per clock cycle, yet an instruction within a software loop may execute arbitrarily many times. To resolve this, the synthesis procedure makes each iteration of a loop execute in a different clock cycle.

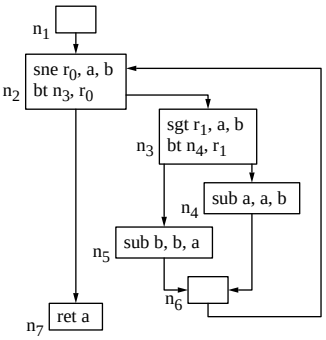
This step introduces arcs that represent control crossing a clock cycle boundary, such as the dashed line from n_8 to n_9 in Figure 1c. This means when control reaches n_8 in a cycle, control begins at n_9 in the next.

We break loops in the CFG by inserting a clock arc along any control path that flows upwards in the program text, such as the implicit branch from the end of a *while* loop to the beginning. We number nodes according to their position in the program text, so any arc from a higher-numbered node to a lower-numbered one is marked. In Figure 1b, only the arc from n_6 to n_2 flows upwards. This may insert more arcs than necessary (solving the minimum feedback arc set problem would give an optimal result), but works fine for human-written code.

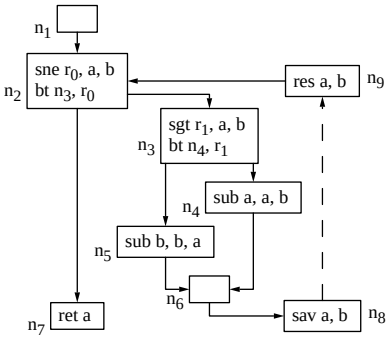
We also insert a clock boundary after each memory write by splitting the node in which the write instruction resides and connecting the two halves with a clock

```
1  int gcd(int a, int b)
2  {
3    while (a != b)
4      if (a > b)
5        a -= b;
6      else
7        b -= a;
8    return a;
9  }
```

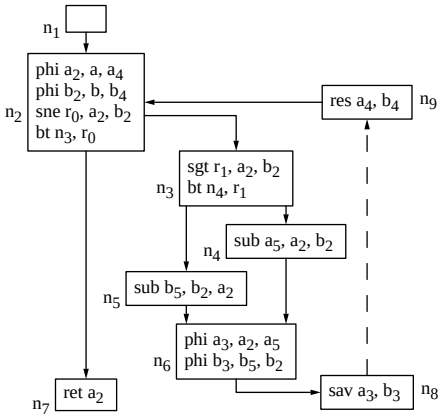
(a)



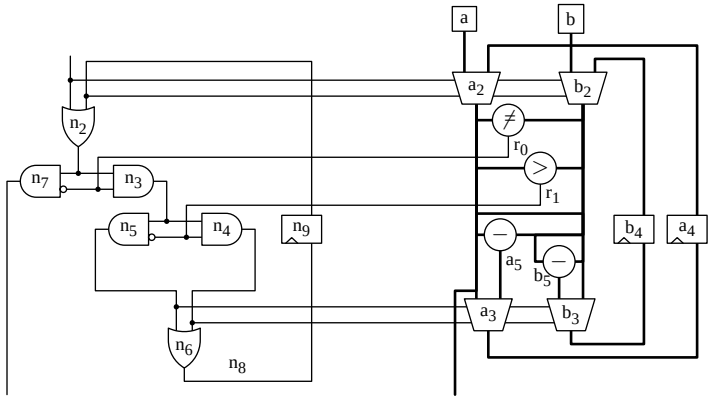
(b)



(c)



(d)



(e)

Figure 1: The synthesis procedure applied to the greatest common divisor algorithm. The function (a) is translated into a control-flow graph (b), clock cycles are identified and save/restore instructions added (c), the program is transformed into static single-assignment form to expose dataflow (d), and the controller and datapath are generated mechanically from the resulting graph (e).

arc. This accommodates memories that prohibit multiple writes within a cycle.

Sav and *res* instructions are inserted to record which variables need to be saved across clock cycles. Not all do. For example, the temporaries r_0 and r_1 are only used within a clock cycle and are always recalculated. We determine which to save using live variable analysis (a classical bit-vector dataflow algorithm [1]).

Since not all variables must be stored every cycle, register assignment could reduce the number of needed registers by storing different variables in the same register. Currently each variable is simply given a unique register.

2.3 Transforming to Static Single-Assignment Form

After adding clock arcs to the CFG, we transform the program to static single-assignment (SSA) form [6] to determine the structure of the datapath. In hardware, a wire takes on exactly one value each cycle, but the same variable may be written multiple times in software. SSA form makes each variable definition unique, adjusts each use of a variable, and adds *phi* functions that selects the relevant definition when control may come from two or more definitions of the same variable.

The SSA transformation works in three steps. First, *phi* functions are inserted where two or more definitions of a variable meet for the first time, such as n_2 in Figure 1c, where the definition of b may come from the beginning of the function or the *res* in n_9. Next, each definition of a variable, including the output of the *phi* functions, is made unique by adding a subscript (e.g., the b assigned in n_5, Figure 1c, becomes b_5 in Figure 1d). Finally, each raw variable is replaced with the most-recently-defined subscripted version. So the b seen by the *sgt* instruction in n_3 is b_2, since b_2 is defined in b_2. The source operands of the *phi* functions reflect the definition visible along each incoming control arc. For example, the b seen in n_6 can be b_5 defined in n_5, or it could be b_2 defined in n_2.

2.4 Generating a Netlist

Once the clock cycles have been identified and the program is in static single-assignment form, generating the controller and datapath is straightforward.

The controller implements the control-flow graph and the datapath implements the arithmetic instructions within each node. Signals from the controller steer data in the datapath using mux select inputs. Predicates tested within the datapath affect how control flows in the controller.

The controller is the left half of the circuit in Figure 1e. One-hot encoding is used throughout. The activation condition for each node is encoded in a wire that takes value 1 when the node is active for the cycle and 0 otherwise. When two or more control arcs enter a node, the activation condition is the logical OR of the arcs from its predecessors, such as n_6. A two-way decision turns into a pair of AND gates, such as n_4 and n_5, that are activated when their node is active (e.g., n_3) and depend on the value of the predicate. The datapath is responsible for computing the value of the predicate, such as r_1.

Each clock arc becomes a latch in the controller: if control reaches the latch at the end of one cycle, control starts from the latch at the beginning of the next.

This procedure generates controllers with some redundancy. In the example, n_6 could be removed and its output connected to n_3 since the two signals are identical. Synthesizing controllers from a control dependence graph [6] would eliminate such redundancy. Also, choosing a different state encoding (i.e., not one-hot) might generate more efficient controllers.

Building the datapath (e.g., the right side of Figure 1e) is straightforward. Each arithmetic instruction becomes an arithmetic subcircuit connected to its operands. For example, the instruction "sg r_1, a_2, b_2" becomes a comparator whose output r_1 is true when its input a_2 is greater than its input b_2. The branch instruction "bt n_3, r_0" in n_2 connects the output of r_0 to the inputs of the n_7 and n_3 AND gates in the controller.

The *phi* functions become multiplexers that steer data through the datapath, in effect selecting which expressions are evaluated (actually, selecting which expressions' results are used). "Phi a_3, a_2, a_5" in n_6 becomes the mux labeled a_3 in Figure 1e, which selects a_5 when control comes from n_4 and a_2 when control comes from n_5.

Although simple in this example, the logic feeding the registers in the datapath can become fairly complicated. In general, which values get latched into each register is a function of which *sav* instructions are active at the end of a cycle. The logic before each register is a mux that selects values based on a set of control signals, each corresponding to a different *sav* node.

2.5 Synthesizing Assert Statements

For each *assert* statement, the synthesis procedure builds a pair of primary outputs: one that is true when the *assert* statement runs and passes, the other true when it runs and fails. The generated circuitry is like that for an *if* statement: the predicate expression is calculated and fed into a pair of AND gates whose outputs are the signals of interest. For example, if `assert(a>b)` were inserted after line 3 in Figure 1a, the synthesis procedure would copy the r_0 comparator and the n_3 and n_7 AND gates.

```
for (i=0 ; i<LENGTH ; ++i)
  a[i] = LENGTH - i + s;

for (j=1 ; j<LENGTH ; ++j) {
  key = a[j]; i = j;
  while (i>0 && a[i-1]>key) {
    --i; a[i+1] = a[i];
   }
  a[i] = key;
}
for (j=1 ; j<LENGTH ; ++j)
  assert(a[j-1] <= a[j]);
```

Figure 2: Insertion sort.

```
    for (j=x[1], i=2 ; i<=LENGTH ; ++i)
      if (x[i]<j) return; else j=x[i];
    for (k=y[1], i=2 ; i<=LENGTH ; ++i)
      if (y[i]<k) return; else k=y[i];

    i = j = k = 1;
M2: if (x[i]<=y[j]) goto M3; else goto M5;
M3: z[k++] = x[i++];
    if (i<=LENGTH) goto M2;
    while (k<=2*LENGTH) z[k++]=y[j++];
    assert(1); return;
M5: z[k++]=y[j++];
    if (j<=LENGTH) goto M2;
    while (k<=2*LENGTH) z[k++]=x[i++];
    assert(1); return;
```

Figure 3: Merge of two sorted lists.

3 Experiments

We tested our approach by using Ketchum to prove some properties and generate test cases for a pair of small functions representative of embedded software. Verifying large systems demands verifying such small pieces.

We expect to find code like insertion sort in embedded software, since it involves a loop, decisions, an array, and simple integer arithmetic. We tried first to prove that the algorithm is correct for a four-element array running with a manually-abstracted three-bit datapath, which took Ketchum over ninety minutes to prove. Of the four states of the last line's *assert* statement, Ketchum spent a few seconds doing random simulation to determine two were reachable, took an hour to prove a third is unreachable using symbolic state-space traversal, and determined the fourth state was unreachable using an abstraction refinement procedure [14] after another half hour.

Next, we tried restricting the input space of insertion sort by adding the first two lines in Figure 2, which force the routine's input to be a decreasing sequence. As expected, this greatly decreased Ketchum's run time because there were fewer cases to consider. We varied the length of the array for the experiments on the left of Table 1, and varied the width of the numbers for those on the right.

Table 1: Time taken to prove the output of the insertion sort algorithm is nondecreasing. E: Length of array to sort, W: Bit width of array elements, L: Number of latches in model *Arbitrary input case; all others run on decreasing input sequences.

E	W	L	Time
4	3	72	97 m*
3	4	40	20 s
4	4	44	41 s
5	4	48	1.4 m
6	4	52	1.5 m
7	4	56	3.3 m
8	4	60	4.0 m
10	4	68	21 m
12	4	76	38 m
15	4	88	159 m

E	W	L	Time
5	5	55	73 s
5	6	62	73 s
5	7	69	73 s
5	8	76	73 s
5	10	90	74 s
5	12	104	71 s
5	16	132	74 s
5	24	188	75 s

Table 2: Time and memory required to generate a test case for the merge example.

Entries	Width	Latches	Time	Memory
6	8	255	11 m	520M
6	10	307	9.6 m	510M
6	14	411	37 m	510M
6	16	463	29 m	520M

The time to verify the restricted insertion sort grows more than quadratically with the number of entries. We expect at least quadratic, since insertion sort itself is quadratic and all of Ketchum's algorithms are roughly linear in the number of cycles required to reach a goal. The additional cost may be due to additional state.

Next, we tried generating test cases for the merge of two sorted lists algorithm taken from Knuth [10, p. 158] shown in Figure 3. The variant we used verifies the two lists are sorted before attempting to merge them. We asked Ketchum to "thread the needle"—to find a test case with two sorted lists where the highest element appears in the second to exercise the second *assert(1)* statement.

Table 2 shows the time it took to find test cases for datapaths of different widths. For all of these cases, the ATPG engine found the input sequence in a time that seems to be growing superlinearly with the size of the design.

These results are worse than expected. Ho et al. [8] were able to classify 65,536 states of a 155-latch design in 75 minutes. But it took nearly twice as long to classify four states in the 72-latch insertion sort example. Clearly, the number of latches is not the sole complexity metric.

4 Conclusions

We sought to determine whether the sophisticated machinery developed for verifying hardware circuits could be applied to software. Our approach as it stands does not seem viable for large programs, but changing the type of program being verified, the synthesis procedure, or the verification approach (i.e., the Ketchum model checker) may improve the situation.

How these programs observe their inputs may be one source of Ketchum's difficulties. A typical hardware circuit samples its inputs once a cycle, whereas the models we are building only check their inputs once in the first cycle. Ketchum tries to steer the model through the state space to goal states by judiciously selecting inputs; this mechanism fails on the software models since everything is determined after the first cycle. Most embedded software does perform I/O, but our procedure is not able to satisfactorily abstract away the intervening behavior.

Increased sequential depth is another challenge of the models we build. In hardware circuits without wide counters, an arbitrary state may be reached within a few cycles, but for something like insertion sort, a reasonable program may take hundreds or thousands of cycles to reach a particular state. Ketchum's engines—especially the bounded ones—work better on shallow properties.

There is room to improve the synthesis procedure, which builds a control-flow graph, identifies clock cycles, transforms the program into static single-assignment form, and mechanically builds a controller and datapath from this representation. Optimizing models by reducing the number of latches or clock cycles may improve things, but probably not by the orders of magnitude we desire.

Adding the automated abstraction of the Bandera tool is an obvious possibility. This would automate the manual datapath abstraction (i.e., reducing the number of bits) we did for the examples and slice away irrelevant parts of the program depending on the property being tested.

Ketchum was not designed to handle the type of models we build for it, and as such finds small examples such as insertion sort far more challenging than bigger hardware examples. One possibility is to attempt to re-engineer some of the verification engines with the different character of these models in mind.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN Model Checking and Software Verification (Proceedings of the 7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, Stanford, California, August 2000. Springer-Verlag.

[3] Thomas Ball and Sriram K. Rajamani. Checking temporal properties of software with Boolean programs. In *Proceedings of the Workshop on Advances in Verification*, Chicago, USA, July 2000.

[4] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–104, 1992.

[5] James Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[8] Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal simulation engines. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 120–126, San Jose, California, November 2000.

[9] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[10] Donald E. Knuth. *The Art of Computer Programming*, volume three. Addison-Wesley, Reading, Massachusetts, second edition, 1997.

[11] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C. In *Proceedings of Design, Automation, and Test in Europe*, pages 312–319, Paris, France, March 2000.

[12] Richard W. Sharp and Alan Mycroft. The FLaSH compiler: Efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Laboratories Cambridge, 2000.

[13] Michael D. Smith and Glenn Holloway. *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*. Harvard University, 2000. Machine SUIF documentation set.

[14] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, Nevada, June 2001.