Partitioned Model Checking from Software Specifications*

Xiushan Feng
Dept. of Computer Science
University of British Columbia
xsfeng@cs.ubc.ca

Alan J. Hu
Dept. of Computer Science
University of British Columbia
ajh@cs.ubc.ca

Jin Yang
Strategic CAD Lab
Intel Corporation
jin.yang@intel.com

Abstract— With the trends toward higher-level design, verification models written in software, and hardware/software codesign, it is increasingly important to verify that RTL hardware behaves correctly according to an executable software specification. In this paper, we propose a natural way to formalize a cycle-accurate software specification as an annotated control flow graph, and then we introduce a novel partitioned model-checking algorithm that exploits the annotated control flow graph. Preliminary experimental results show that our new method runs faster than standard model checking.

I. Introduction

It is increasingly important to be able to verify an RTL (or similarly low-level) design against a higher-level specification written in software. For example, many designs are now starting in C or C-like languages, meaning that there is a need for C-to-RTL verification, just as RTL-to-gate verification become important with the widespread adoption of logic synthesis. Another example occurs in hardware/software codesign, where we may wish to migrate part of a design from software to hardware, so we must verify hardware against a software specification. And in functional verification, increasingly companies are writing high-level functional models in software, to allow much faster preliminary verification, but then there remains the problem of verifying that the RTL hardware behaves the same as the software verification model.

Some pioneers have already been researching the verification of hardware versus software. If the software is register-transfer level, the problem is basically RTL-to-RTL verification, and commercial tools can be used [12]. If the software is arbitrary, high-level code, then full formal verification is undecidable, but bounded-length verification is possible using symbolic execution [8, 2, 3].

We consider an intermediate case: software that is higher-level than RTL, but is still cycle and pin accurate. This means that the software model can use different implementations than the RTL hardware (e.g., we could program a priority queue easily using normal software, rather than building it in RTL hardware) and doesn't have to be synthesizable. However, we assume that the user specifies bit-accurate inputs and outputs of the block, and the software model must produce the correct outputs on the correct clock cycles. Such software specifications have started to be used in industry for high-end processors and system designs [7].

We believe that this intermediate level of software is high-level enough to be helpful for specifying hardware, and that the restrictions on the software make formal verification much simpler than for arbitrary software. In this paper, we present a natural way to formalize these software specifications as annotated control flow graphs, and we introduce a novel, partitioned, full (unbounded) model-checking algorithm that exploits the graphs.[1]

II. Annotated Control Flow Graphs

In order to make our work widely applicable to any software language, we have created a simple, abstract model for describing a specification written in software. This decouples our research from the time-consuming-but-well-known work of creating specific language parsers. The model is based on control flow graphs, a standard concept from software analysis.

A. Control Flow Graphs

A control flow graph (CFG) is a graph G(V,E), with V the set of vertices, and E the set of edges. Each vertex is a block of straight-line code (a basic block), and an edge is a jump in the control flow. Multiple outgoing edges from a vertex indicates conditional branching. CFGs are a standard way to describe the behavior of software, representing all alternatives of control flow.

B. Annotating the Control Flow Graph

In this paper, we introduce a modified control flow graph, which we call an *annotated control flow graph* (ACFG). We were inspired by the efficiency of *Generalized Symbolic Trajectory Evaluation* (GSTE) [13], so ACFGs are based on the assertion graphs used in GSTE. GSTE was developed at Intel

*This work was supported in part by grants from Intel Corporation and the Natural Science and Engineering Research Council of Canada.

[1] Thanks to the anonymous reviewers, we have learned of very recent work [5] that also considers unbounded model checking of hardware versus software. The strengths of that work are powerful abstraction techniques to reduce the complexity of the software specifications as well as a feature-rich language front-end; they use standard symbolic model checking for the verification. In contrast, we deliberately avoid focusing on the language front-end, and concentrate on improving the core model checking. We believe abstraction and improved model checking will both be necessary in practice.

and has proven to be a success on leading-edge designs due to its superior efficiency and capacity (e.g., [1, 9]).

GSTE extends classical symbolic trajectory evaluation [11] by using an assertion graph to define the property to be verified. An assertion graph is a kind of automaton that has similar expressive power to ordinary model checking [4, 6]. GSTE theory gives an efficient procedure to do model checking that a hardware design obeys an assertion graph. Details on the theory of GSTE and assertion graphs can be found elsewhere [13].

However, GSTE is fundamentally an approach for circuit verification. It doesn't handle complicated software/hardware verification, because assertion graphs were not designed to model software. For example, assertion graphs contain "symbolic constants" that are used somewhat like local variables, but the value of a symbolic constant does not change, whereas the basic idea of software and CFGs is the assignment of new values to variables — the assertion graph is declarative, whereas software is imperative. Furthermore, there are no internal variables in an assertion graph; the circuit and the assertion graph share all the same state variables. For a specification written in software, the circuit and the specification need to have their own, separate variables.

To solve these problems, we combine ideas from CFGs and GSTE assertion graphs to form our ACFG structure. As in GSTE assertion graphs, the edges are labeled, rather than the vertices, as in CFGs. However, paths through an ACFG, like paths through a standard CFG, correspond to control flow through a program. The labels include antecedents and consequents, as in GSTE assertion graphs. However, we also introduce *assignments*, which assign new values to variables.

For example, Figure 1 shows a simple ACFG which describes a counter circuit that counts up to 4. The ACFG has a 1-bit input `load`, a 3-bit input `D`, and a 3-bit state variable and output `X`. (The circuit being verified is assumed to have the same I/O signals, but it's internal state might be different. The user declares which variables are inputs, outputs, and state variables, as well as the correspondence between ACFG and circuit signals.) Each edge label has three parts: antecedent/assignment/consequent. The concept of antecedents and consequents come from GSTE: they are boolean-valued formulas over the variables. Intuitively, the antecedent is a condition that must be true in order to follow that edge, and the consequent is a condition that the verification tool must check if the computation follows that edge. The assignment computes values for the variables and updates them. The example ACFG loads a 3-bit number, saves it to the internal variables $X[0..2]$ at the entry vertex (vertex 0). $X[0..2]$ are also output variables. After the number is loaded, the control flow of the ACFG will jump to vertex 1, and branch based on the value of the number. If the number is less than 4, it increases by one for each cycle until it hits 4. If the current counter output is equal or greater than 4, it holds that value for consequent cycles on the terminate vertex (vertex 2 on the graph).

More formally, an ACFG is an edge-labeled, directed graph, with a distinguished initial vertex v_I. Each edge e is labeled with an antecedent $ant(e)$, an assignment $assi(e)$, and a consequent $cons(e)$. Both $ant(e)$ and $cons(e)$ are propositional formulas over inputs and state variables. The assignment $assi(e)$ associates a formula over the current variables to each state variable. (For variables that do not change value on an edge, the formula is simply the variable itself.) When an assignment is executed, the formulas are evaluated using the current values of the variables, and then the variables are updated with the value of the associated formula. When the flow of control is at a vertex v_i, an edge e ($v_i \rightarrow v_j$) can execute the assignment $assi(e)$ iff it satisfies the ant(e). The consequent $cons(e)$ is a property that is expected to hold after the $assi(e)$ execution.

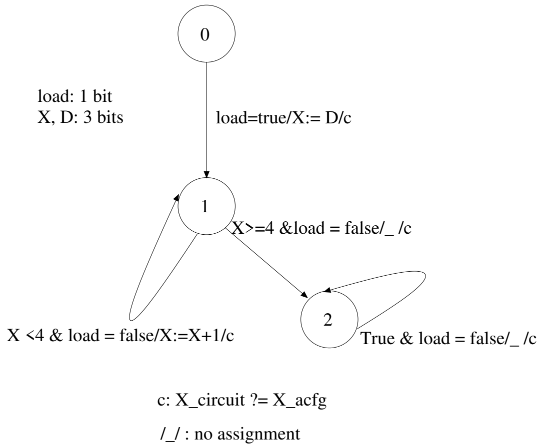

Fig. 1. ACFG for a Counter circuit

In order to map the cycle-accurate-execution of circuits to ACFGs, we define that each edge of the ACFG takes one clock cycle. For convenience, we allow a sequence of assignment statements on each edge, but the entire sequence is executed in one clock cycle. In general, any software that can be reduced to static single assignment (SSA) form can be put on an edge label (e.g., if-then-else, loops that can be completely unrolled, etc.).

On the graph, only paths starting from the entry vertex are considered. Each label describes a possible state of the system in one time step, which corresponds to one clock cycle in the circuit being verified. A trace of the circuit under verification obeys the property specified by the ACFG graph if and only if on every finite path of the same length as the trace, if the antecedents on all edges are satisfied, the consequents are satisfied as well. The verification task is to model check that all traces of the circuit obey the ACFG.

III. Partitioned Model Checking

We start the section by first defining a simple computation model. Let I_g, S_g, O_g be the sets of boolean input, state, and output variables in the ACFG. Similarly, let I_c, S_c and O_c be those in the implementation circuit model. For simplicity, we assume that there is a one-to-one mapping between the interface (i.e., input, output variables) of the high level model captured by the ACFG and the interface of the circuit model. A state in a model m ($m = g,c$) is simply a boolean assignment to the input and state variables $V_m = I_m \cup S_m$ in the model.

For the ACFG, the set of assignments $assi(e)$ on an each edge e defines a state transition relation in the high level model associated with e. From now on, we shall denote such a transition relation as $assi(e)(V_g, S'_g)$ where S'_g is a copy of S_g for holding the values after the transition. For the circuit model, let $R_c(V_c, S'_c)$ be the single state transition relation. Further, let $F_m(o_m)(V_m)$ be the output function for variable $o \in O_m$ ($m = g,c$).

To model check the circuit implementation against the ACFG, the standard approach is to build a single product automaton combining the ACFG and the circuit model, and then to perform reachability analysis on the combined model. However, such an approach is prone to the state explosion problem.

Our model checking solution, instead, is based on GSTE-style model checking proposed in [13]. The key idea is to partition the software states and the circuit states based on the structure of the ACFG, and use the flow and the antecedents in the ACFG to guide and tailor the state space exploration. More specifically, we would like to compute a simulation relation $sim(e)(V_g, V_c)$ for each edge e in the ACFG, such that the simulation relation tells us (1) what is the set of states the software model can be in on the particular edge e, and (2) for each such state, what is the corresponding set of possible states in the circuit model. The consequent on the edge is then checked against the simulation relation.

Before describing the model checking algorithm, let us first define two partial post-image transformers. Given a global state predicate $P(V_g, V_c)$, the partial post-image transformer of the predicate with respect to an edge e in ACFG is:

$$\begin{aligned} &post(e)(P(V_g,V_c)) = \\ &\quad (\exists V_g . P(V_g,V_c) \wedge assi(e)(V_g,S'_g))[S'_g/S_g] \end{aligned}$$

where S'_g/S_g denotes the substitution of every variable v' in copy S'_g with its original variable v in S_g. Similarly, the partial post-image transformer of the predicate with respect to the circuit is

$$\begin{aligned} &post_c(P(V_g,V_c)) = \\ &\quad (\exists V_c . P(V_g,V_c) \wedge R_c(V_c,S'_c))[S'_c/S_c]. \end{aligned}$$

Figure 2 lists our GSTE style model checking algorithm in its entirety. Line 3 applies antecedents from the ACFG to the circuit being verified. The rest of the lines 2–10 simply initialize the computation, with initial task queue entries from the initial vertex. The main verification loop in lines 11–22 propagates newly discovered relations between graph edges and circuit states in an event-driven manner, until a fixpoint is reached (or a bug is found). The typical $cons(e)$ checked in line 13 would be that the corresponding outputs agree, but more general properties can be verified. The partitioning occurs because we compute separate BDDs for each $sim(e)$.

Because the algorithm is very similar to GSTE-model checking, we hope in the future to exploit GSTE-style abstraction for higher capacity.

1: **function** ModelCheck($ACFG$, $post_c$)
(* Incorporate mapped antecedents to the circuit and initialize the simulation relation *)
2: **for** all edges e of $ACFG$ graph **do**
3: $\quad ant(e) := ant(e) \wedge ant(e)[I_g/I_c][O_g/O_c]$;
4: $\quad$ **if** e is from the entry vertex **then**
5: $\quad\quad sim(e) := post(e)(ant(e))$;
6: $\quad\quad$ add e into taskQueue;
7: $\quad$ **else**
8: $\quad\quad sim(e) := \emptyset$;
9: $\quad$ **end if**
10: **end for**
(* Compute simulation relation and check consequents *)
11: **while** taskQueue $\neq \emptyset$ **do**
12: $\quad$ remove an edge e from taskQueue;
13: $\quad$ **if** $sim(e) \not\Longrightarrow cons(e)$ **then**
14: $\quad\quad$ return(a counter-example trace);
15: $\quad$ **end if**
16: $\quad$ **for** each successor edge e' of e **do**
17: $\quad\quad sim(e') := sim(e') \vee post(e)(post_c(sim(e)) \wedge ant(e'))$;
18: $\quad\quad$ **if** there is a change in $sim(e')$ **then**
19: $\quad\quad\quad$ put e' into taskQueue;
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **end while**
23: return(succeed);

Fig. 2. A partitioned model checking algorithm for ACFG

function SRT_Division($dividend, divisor, N$)
$P^0 = dividend$;
for $i = 1$ **to** N **do**
$\quad$ **if** $2P^{i-1} \geq 0.5$ **then**
$\quad\quad q_{N-i} = 1$;
$\quad\quad P^i = 2P^{i-1} - divisor$;
$\quad$ **else**
$\quad\quad$ **if** $2P^{i-1} < -0.5$ **then**
$\quad\quad\quad q_{N-i} = -1$;
$\quad\quad\quad P^i = 2P^{i-1} + divisor$;
$\quad\quad$ **else**
$\quad\quad\quad q_{N-i} = 0$;
$\quad\quad\quad P^i = 2P^{i-1}$;
$\quad\quad$ **end if**
$\quad$ **end if**
end for
if $P^N < 0$ **then**
$\quad P^N = P^N + divisor$;
$\quad q = q - 1$;
end if

Fig. 3. SRT Unsigned Fractional Division algorithm

	Without Partitioning			ACFG Partitioned Model Checking			
N	Time(s)	BDD size	% reordering time	Time(s)	BDD size	% reordering time	Speedup
7	12.27	5390	95.2%	6.99	7462	94.4%	1.76x
8	32.31	10307	95.7%	18.41	14635	95.6%	1.76x
9	21.89	8982	95.8%	14.97	9770	96.6%	1.46x
10	50.62	6619	74.9%	26.26	15566	93.1%	1.93x
11	392.66	10915	98.3%	379.52	59125	97.8%	1.03x
12	727.43	19722	98.1%	484.12	29792	98.0%	1.50x
13	1854.62	63555	95.9%	877.29	38756	97.1%	2.11x
14	950.25	22262	95.9%	633.86	44919	96.2%	1.50x
15	452.57	20036	97.6%	193.19	56982	96.4%	2.34x

TABLE I
STANDARD, UNPARTITIONED MODEL CHECKING VS. ACFG PARTITIONED MODEL CHECKING. RESULTS USING DYNAMIC VARIABLE REORDERING.

IV. EXPERIMENTAL RESULTS

We have conducted experiments to test the performance of our model-checking algorithm. As our test case, we use a radix-2 SRT unsigned fractional division circuit. Fig. 3 shows the algorithm. There are several reasons for choosing this circuit. One reason is that the SRT algorithm is well-known and widely used in practice, and the circuit is fairly small, but the specification still has for-loops and if-then-else branches. Modeling such kinds of control flow is the goal of using ACFGs. Another reason is that the example is scalable, so we can test the capability of our verification tool. We have a parameterized circuit generator tool that builds a divider circuit with $2N$-bit dividend and N-bit divisor for any N. As is done in practice, the circuit uses a redundant representation for the partial remainder to avoid needing a long carry chain, which is the advantage of SRT division.

By using an ACFG to describe the algorithm, we can have local variables for the specification, and a flexible control-flow graph structure to describe the control flow. We define the single-cycle operations on each edge. The operations are an ordered sequence of assignments. As noted earlier, these sequences are just convenience to make it is easy to define a complicated, single-cycle operation without writing a hard-to-understand single-assignment. The key idea is that the specification model should be easy to maintain and change.

The following is an example assignment sequence for an edge of the SRT specification. P is the partial remainder. qp is the bit to record that the next quotient bit gets an assignment 1; qn is the case for -1. After each cycle, they all shift left by one bit.

```
-->P[2*N-1..N-1] :=
   P[2*N-1.. N-1] - [0]:D
-->qp := 2*qp
   && qn := 2*qn
   && P:=2*P
-->qp := qp xor 1
-->P[2*N] ==0
   && remainder := P[2*N-1..N]
   && quotient := qp - qn
  ||
   P[2*N] ==1
   && remainder := P[2*N-1..N] + D
   && quotient := qp - qn - 1
```

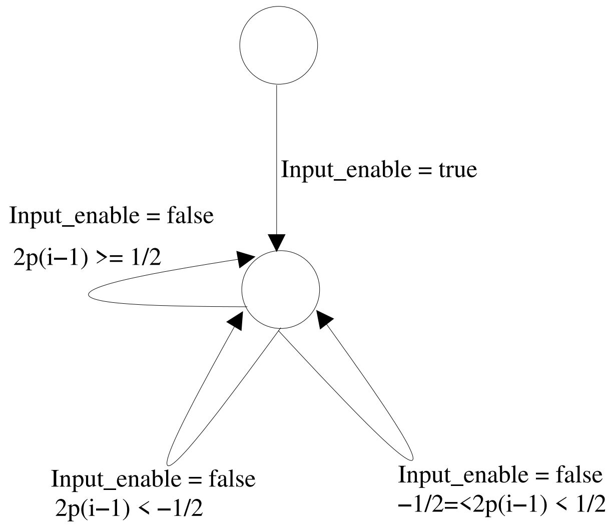

Fig. 4. ACFG for SRT Algorithm. For brevity, only antecedents are shown.

There are four ordered steps in this sequence assignment, indicated by the --> operator. Inside each step, the order of assignments doesn't matter. Step 4 has predicated assignments. If the sign bit of P is 0 (positive),

```
remainder := P[2*N-1..N]
  && quotient := qp - qn
```

will be executed. If not, then

```
remainder := P[2*N-1..N] + D
  && quotient := qp - qn - 1
```

will be executed.

Fig. 4 sketches the main part of the specification ACFG. The three self-loop edges correspond to the 3-way branch structure in the loop on lines 3–16 of Fig. 3.

The remainder and quotient are the outputs. They are the final results after N+1 cycle. However, because the specification is cycle-accurate, we check the intermediate results for each cycle by using the consequents in the ACFG.

All experiments were run on a 2.6Ghz Pentium 4 with 4GB of RAM. When running with a reasonable BDD variable order, run time rather than memory was always the limiting factor. BDD-based model checking is extremely sensitive to the variable order. We have not found an obviously good variable order, and we feared that we might inadvertently bias our experiments if we chose a static order that is better for one method over the other. Instead, we ran our experiments with dynamic BDD variable reordering enabled, using CUDD_REORDER_SIFT in the CUDD BDD package. (We tried other methods, with similar results.) The initial order was arbitrary, except that we put the variables that symbolically encode the graph (in the standard method without partitioning) at the top of the order. Doing so should make the initial BDD sizes similar for the standard method and our new partitioned method, so the comparison should be fair.[2]

Table I shows results for this experiment. BDD sizes varied, but were generally similar between the two methods, as expected. However, in all cases, the new partitioned model checking approach ran faster, sometimes must faster. Strangely, sometimes larger instances required less time and/or smaller BDDs. We believe this is due to dynamic reordering, and note that the dynamic reordering time dominated total runtime.

As an aside, we note that GSTE-style pruning using the antecedents is very helpful. If we complete the graph in Fig. 4 with all the edges for when the antecedents are false as well as true, as would be needed in a standard CFG, the model checking is much more expensive.

V. Conclusion And Future Work

We have presented a method to model cycle-accurate software using annotated control flow graphs and introduced a novel, partitioned model checking algorithm for verifying RTL hardware against cycle-accurate software. Our empirical results show that our ACFG model checker allows faster verification.

This work is a first step towards the general problem of verifying RTL hardware against higher-level, but cycle-accurate, specifications. The next step will be handling more general control flow (such as loops, procedure calls, etc.) within a single clock cycle. The other main direction for future work is to improve efficiency for handling larger problems.

[2] In a recent paper presented at CAV 2004, Sebastiani et al. presented impressive results showing an exponential improvement in memory usage for GSTE-style partitioned model checking applied to conventional LTL model-checking [10]. Discussion at the conference, however, indicated that most or all of the memory usage improvement would disappear with a variable order that put at the top the BDD variables that symbolically encode the graph. Our work partitions the model checking similarly to theirs, so we must be very careful to incorporate this understanding.

References

[1] Bob Bentley. High level validation of next generation microprocessors. *Intl Workshop on High-Level Design, Validation, and Test*. 2002.

[2] Edmund Clarke and Daniel Kroening. Behavior consistency of C and Verilog programs using bounded model checking. Tech Rep CMU-CS-03-126, Carnegie Mellon, May 2003.

[3] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. *Asia South-Pacific Design Automation Conf*, pages 308–311. 2003.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logics of Programs*, pages 52–71, May 1981. LNCS 131.

[5] Himanshu Jain, Daniel Kroening, and Edmund Clarke. Verification of SpecC using predicate abstraction. *Intl Conf on Formal Methods and Models for Codesign*. 2004.

[6] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. *5th Intl Symp on Programming*, pages 337–351. 1981. LNCS 137.

[7] Wolfgang Roesner. What is beyond the RTL horizon for microprocessor and system design? Invited talk at the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2003, October 2003.

[8] Hiroshi Saito, Takaya Ogawa, Thanyapat Sakunkonchak, Masahiro Fujita, and Takashi Nanya. An equivalence checking methodology for hardware oriented C-based specifications. *Intl High-Level Design, Validation, and Test Workshop*, pages 139–144. 2002.

[9] Tom Schubert. High level formal verification of next-generation microprocessors. *Design Automation Conf*, pages 1–6. 2003.

[10] Roberto Sebastiani, Eli Singerman, Stefano Tonetta, and Moshe Y. Vardi. GSTE is partitioned model checking. *Computer-Aided Verification: 16th Intl Conf*. 2004. LNCS 3114.

[11] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, 1995.

[12] Luc Séméria, Andrew Seawright, Renu Mehra, Daniel Ng, Arjuna Ekanayake, and Barry Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. *Design Automation Conf*, pages 123–128. 2002.

[13] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *Intl Conf on Computer Design*, pages 360–365. 2001.