Structured Formal Verification of a Fragment of the IBM 390 Clock Chip (Extended Abstract)

Alfons GESER[*]
WSI, U Tübingen

Wolfgang KÜCHLIN[*]
WSI, U Tübingen

We present a simple and powerful method for formal verification of hardware, with stress on the use of symmetries. We focus on gate-level descriptions of hardware which are given in the form of a *netlist*. Signals are modelled by propositional logic. In our approach the verification problem breaks down in three steps:

1. Compilation of the netlist into modules of a typed term graph rewriting system. Sequential behaviour is modelled by deterministic Mealy automata.
2. The formal requirements are rewritten into a unique normal form, which is a term graph representing a propositional formula.
3. The propositional formula is evaluated as a functional decision diagram [8, 7]. If the result represents 1 then the answer is "yes", else a counterexample may be constructed mechanically on request.

We illustrate these steps at a case study: a fragment of the IBM 390 Clock Chip [11, Section 2.8].

1 Netlists and Their Compilation to Term Graph Rewriting Systems

It pays to have at one's disposal a language wider than propositional logic in order to express hardware behaviour. We chose *term graph rewriting* [10, 5] for two reasons: Like term rewriting [6] it is expressive and easy to use, moreover the structure of hardware can be mapped faithfully into term graphs.

Recall that a *term graph structure* is a set of nodes, a set of arrows between nodes, and a set of function symbols, such that the following properties hold. Each function symbol is assigned a fixed nonnegative integer, its arity. A node may be labelled by a function symbol, otherwise it is called a *variable node.* The number of incoming arrows to a node must equal 0 for a variable node, and must equal the arity of the label of the node otherwise.

A *term graph* now is a node together with a term graph structure; a term graph rewriting rule is a pair of nodes together with a term graph structure.

The fact that a node o is labelled by f and has incoming arrows from nodes $i_1, \dots, i_n$ may be expressed by an *assignment* $o = f(i_1, \dots, i_n)$. So a set of assignments forms a term graph structure.

[*]Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, {geser,kuechlin}@informatik.uni-tuebingen.de

The translation from netlists to term graphs is quite natural: A *net* (i.e. a point that carries a signal) in the hardware translates to a node in the term graph; an output net o of a *box* (i.e. a primitive component) whose inputs are the nodes $i_1, \ldots, i_n$ translates to an assignment $o = f(i_1, \ldots, i_n)$ where f is a symbol for the Boolean function realized by the box output.

For instance assume that one wants to model the AND by two NANDs. The meaning of $\mathtt{and}(x, y)$ is adequately given by the term graph rewriting rule

$$\mathtt{and}(x,y) \to \mathtt{nand}(z,z) \quad \text{where} \quad z = \mathtt{nand}(x,y),$$

as opposed to the term rewriting rule $\mathtt{and}(x,y) \to \mathtt{nand}(\mathtt{nand}(x,y), \mathtt{nand}(x,y))$ which may also be viewed as a term graph rewriting rule but which needs 3 nodes labelled by "nand" rather than 2.

As one assignment is created per box output, the size of the created term graph is obviously linear in the size of the hardware. This is in contrast to proper term rewriting where the created term may be of exponential size. Linearity is most important in view of big circuits. For instance our fragment has approximately 580 nets and 170 flipflops.

2 Compilation of Combinatorial and Sequential Hardware

Combinatorial hardware is modelled as usual by Boolean vector functions which are specified by constructor based, explicit term graph rewriting rules. "Explicit" here says that $0, 1$ must not occur at the left hand side of a rewrite rule. Explicitness preserves expressive power since every propositional function can be represented by a propositional formula.

To model sequential behaviour of hardware, one has to introduce a form of discrete timing. We stick to the well-known concept of *deterministic finite-state machine*, more accurately its *Mealy automaton* variant, to model sequential behaviour of hardware. Recall that a Mealy automaton is given by three finite sets, S, I, O, whose elements are called states, inputs, and outputs, respectively; an element $s_{\mathtt{init}} \in S$ called the initial state; a function $\delta : S \times I \to S$, the step function; and a function $\lambda : S \times I \to O$, the output function.

We encode states, inputs, and outputs each as Boolean vectors. The pins of the chip become the inputs and outputs of the state machine. The storing elements of the chip are D-type flipflops. Assuming all flipflops numbered consecutively, the present state of the chip is given by the enumeration of all nets incident with flipflop Q outputs; the subsequent state by the enumeration of their D inputs. So the state transition function is virtually just another name for the combinatorial part of the state machine, and so easily specified as a term graph rewriting rule.

For instance a two-bit counter (with no inputs) is specified as follows:

$$\begin{gathered}\delta(\mathtt{mk}_S(q_1, q_0), \mathtt{mk}_I) \to \mathtt{mk}_S(d_1, d_0) \\ \text{where} \quad d_0 = \mathtt{sum}(q_0, 0, 1), \quad c_1 = \mathtt{carry}(q_0, 1, 0), \quad d_1 = \mathtt{sum}(q_1, 0, c_1)\end{gathered}$$

Here $\mathtt{mk}_S, \mathtt{mk}_I$ are the function symbols that construct Boolean vectors for states, inputs, respectively. $\mathtt{sum}$, $\mathtt{carry}$ denote auxiliary functions for the sum and carry bits, respectively, of a full adder circuit that may be defined by appropriate rewriting rules.

3 FDD evaluation

A requirement is formalized as a Boolean valued term graph which contains no free variables but of type Boolean. Auxiliary function symbols of any type may be introduced by rewriting rules, and used freely within formal requirements. In a first pass, the requirement is rewritten by the given term graph rewriting rules to a *normal form*, i.e. a term graph that cannot be rewritten further.

By the explicit form of rewrite rules, the normal form is a term that consists only of propositional variables and functions. Thus the problem has shrunk to a propositional validity problem, which is solved in the second pass of the procedure. Numerous methods that solve the validity problem are known, e.g. Davis-Putnam (see e.g. [13]) or binary decision diagrams [2]. We use a naive implementation of *functional decision diagrams* [8, 7]. As term rewriting, when applied to Boolean connectives, performs much worse than propositional solvers, we exclude the respective term rewriting rules from the rewriting process.

4 Case Study

The fragment of the chip that we verified is concerned with the collection of set and reset pulses in order to get a synchronized stop signal for a set of chips that constitute the 390 microprocessor.

We are given an informal requirement stating that after a pulse of length one on any of some 22 lines, each of some 30 output lines goes HIGH; after a pulse of length two it goes LOW.

A netlist of the chip fragment is given in IBM's netlist language VIM, by means of the usual propositional connectives and D-type flipflops. In the given netlist there are moreover multiple clocks and bi-directional lines which we drop to simplify our presentation.

Inspection of the chip fragment shows that for each input there is an identical stage that determines whether a pulse of length one or two has been issued at this input. The results are OR-ed for all inputs; they yield set and reset signals, respectively, for a RS type flipflop. Each output of the fragment is preceded by a separate flipflop that buffers the output of the RS type flipflop.

Intuitively it is fairly clear that the specification should hold. The main problem with the verification is complexity caused by repetition. Let us now briefly discuss how the requirements can be formalized in our framework.

For instance one may claim that each of the 30 outputs of the chip carries the same signal. Then by symmetry the remaining propositions need only speak about one such output.

In fact the proposition is not true for *all* states: If the buffering flipflops for the various outputs carry different values then the proposition is violated. Note that the common signal must pass through the buffering flipflop before it arrives at the corresponding output. But the proposition is true for all states that are *reachable* by transitions from the initial state.

It is known that reachability can be expressed as a fixpoint. But how can one quantify over all reachable states in a logic as poor as ours? We prove the proposition for the initial state and show that its validity for an arbitrary state entails its validity for the next state.

Given that P is a proposition on states these can be encoded as

$$P(s_{\mathtt{init}}) * (P(s) \Rightarrow P(\delta(s,i))) \qquad \text{where}$$
$$s = \mathtt{mk}_S(s_{n-1}, s_{n-2}, \ldots, s_0), \qquad i = \mathtt{mk}_I(i_{m-1}, i_{m-2}, \ldots, i_0)$$

where $*$ is the infix symbol for AND.

It is more difficult to prove symmetry in the inputs since there is only *equivalence* up to renaming of inputs. We pursued the following way. First we specify for each input number $1 \leq k \leq m-1$ a map $\phi_k : S \rightarrow S$ which we intend to describe a bisimulation of the given machine with the machine where the first and the k-th input are exchanged. Intuitively, to each flipflop affected at some moment by the first input we assign the flipflop affected at that moment by the k-th input. Admittedly this requires some inspection. With that we claim bisimulation and equality of outputs:

$$(s_{\mathtt{init}} =_S \phi_k(s_{\mathtt{init}})) * (\phi_k(\delta(s,i)) =_S \delta(\phi_k(s), i')) * (\lambda(s,i) =_O \lambda(\phi_k(s), i'))$$
$$\text{where}$$
$$s = \mathtt{mk}_S(s_{n-1}, s_{n-2}, \ldots, s_0), \qquad i = \mathtt{mk}_I(i_{m-1}, i_{m-2}, \ldots, i_0),$$
$$i' = \mathtt{mk}_I(i_{m-1}, \ldots, i_{k+1}, i_0, i_{k-1}, \ldots, i_1, i_k)$$

Here bitwise equality of states and of outputs are assumed specified as Boolean functions $=_S, =_O$, respectively, in the usual way.

In the same spirit one may design the claim that the sequence 010 (011) on the first input yields a set (reset) signal, and the claim that the absence of such a sequence on each input yields no set (reset) signal.

We have made tests for each proposition and obtained results in the order of magnitude of a minute on a SPARC 10 workstation. It turned out that for a realistic FDD computation one needs plenty of RAM.

5 Conclusion and Related Work

Our approach to hardware verification requires only standard knowledge in term graph rewriting, propositional reasoning, and automata theory. In particular, no knowledge in temporal logic is needed. We have shown that in spite of its weak expressive power this framework supports symmetry reasoning and safety properties.

The work reported here is an extension of Bündgen/Küchlin's method to verify the Sparrow processor [3, 4], and its application to industrial hardware. We carry over the spirit, the plan, and much of the experience of this work. The following are new:

1. the rigorous use of term graph rewriting,

2. the encoding of sequential behaviour by a state transition function,

3. the two-step rewriting/evaluation approach.

Among the implemented systems that also combine rewriting with built-in propositional reasoning we are aware of PVS [9] and NQTHM [1].

References

[1] Robert S. Boyer and J. Strother Moore. Proof-checking, theorem-proving, and program verification. *Contemporary Mathematics*, 29:119–132, 1984.

[2] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Comp. Surveys*, 24(3), Sept. 1992.

[3] Reinhard Bündgen and Wolfgang Küchlin. Term rewriting as a tool for hardware and software codesign. In Jerzy Rosenblit and Klaus Buchenrieder, editors, *Codesign — Computer-Aided Software/Hardware Engineering*, pages 19–40. IEEE Press, 1995.

[4] Reinhard Bündgen and Wolfgang Küchlin. Verification of the Sparrow processor. In *ECBS'96*. IEEE Press, 1996.

[5] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In van Leeuwen [12], chapter 5, pages 193–242.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In van Leeuwen [12], chapter 6, pages 243–320.

[7] Udo Kebschull. *Verhaltensbasierte und spektrale Logiksynthese mehrstufiger Schaltnetze unter Verwendung von Binärbäumen.* PhD thesis, Universität Tübingen, D, June 1994.

[8] Udo Kebschull, Endric Schubert, and Wolfgang Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Design Automation Conference (EURO-DAC)*, 1992.

[9] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. 8th Conf. Computer Aided Verification (CAV)*, LNCS 1102, pages 411–414. Springer, 1996.

[10] M. Ronan Sleep, M. J. Plasmeijer, and Marko C. J. D. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice.* John Wiley & Sons, 1993.

[11] Wilhelm G. Spruth. *The design of a microprocessor.* Springer, 1989.

[12] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics). Elsevier - The MIT Press, paperback edition, 1994.

[13] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symbolic Computation*, 21:543–560, 1996.