

Integrating Design and Verification Environments Through A Logic Supporting Hardware Diagrams*

Kathi Fisler[†] and Steven D. Johnson
Department of Computer Science
Lindley Hall 215
Indiana University
Bloomington, IN 47405
{kfisler, sjohnson}@cs.indiana.edu

Abstract — Formal methods and verification tools are difficult for designers to use. Research has been concentrated on handling large proofs; meanwhile, insufficient attention has been paid to the reasoning process. We argue that a heterogeneous logic supporting hardware diagrams and sentential logic provides a natural framework for reasoning and for the formal integration of design and verification environments. We present such a logic and demonstrate its flexibility on fragments of a traffic light controller design and verification problem.

I. INTRODUCTION

Although formal verification is gaining acceptance as a useful and viable stage in the hardware design cycle, designers are not adopting verification as readily as researchers might have expected or hoped. Reasons cited include concern that verification will lengthen the design cycle, the general reluctance of industry to risk applying new techniques, and designers' lack of experience using the formal notations underlying verification systems [14]. These are all valid concerns, but the last point is perhaps the most telling; verification tools that were natural to use would reduce the problems associated with inexperience, which would in turn diminish the first concern.

Most researchers accept that there is considerable learning overhead associated with applying formal techniques; people often suggest that highly trained verification specialists are required to assist designers in using the tools [14]. This state of affairs, although undesirable, is currently accepted as inevitable. However, the correction of a crucial, but subtle, oversight in the research community stands to alleviate the problem.

The community has made the mistake of equating proof and reasoning. When promoting formal methods, we focus on the end result — a verified product. We do not give much attention to the means by which we arrive at

that product, so long as those means are formal and rigorous. This is not sufficient. Verification is about proof, but in practice it requires human reasoning. As a result, the tools and theories used must be conducive to clear reasoning. Our current tools are conducive to proof; support for proof was among the main arguments originally given in favor of doing design using formal logics. We must now develop tools and theories that are also conducive to reasoning.

In this paper, we demonstrate a design and verification logic engineered to support reasoning through its integration of multiple representations of hardware, some of them diagrammatic. Section II explores the nature of reasoning in hardware design. Section III presents examples of hardware reasoning with diagrams that are supported in the logic. Conclusions and future directions appear in section IV.

II. REASONING AND HARDWARE DESIGN

Barwise and Etchemendy view valid reasoning as “the exploration of a space of possibilities” [3]; certainly this view seems consistent with what occurs during the design process, where design alternatives are explored to arrive at an implementation. They note that reasoning is a *heterogeneous* activity — people use multiple representations of information while reasoning, and those representations are often non-sentential forms such as diagrams. This is also consistent with what occurs during design, in which a combination of state machines, circuit diagrams, timing diagrams, and sentential languages such as VHDL are often used.

How well do the representations used in current tools and methodologies support reasoning as described above? In order to support exploration of a space of possibilities, a representation must make explicit the information relevant to the reasoning task at hand; the relevant information often changes depending upon the characteristic being examined. In the case of hardware design, we are sometimes interested in control flow, sometimes functional behavior, sometimes low-level timing. Some representa-

*This paper was revised after distribution of the participant's proceedings to correct errors in figure 3.

[†]Research supported by the AT&T PhD Fellowship Program.

tions support certain characteristics nicely; automata, for example, are good models of control behavior.

Higher-order logic has long been promoted as a good representation for specifying hardware [11]. Control flow and functional behavior can be expressed using higher-order logic. If we are interested in functional behavior, higher-order logic leaves connections between data implicit through the use of variables. Someone using a higher-order logic specification to track data flow needs to mentally connect the sentences involving common variables. Contrast this with the same specification viewed as a circuit diagram, in which the connections are made explicit and the user has no need to mentally trace connections. In this case, the circuit diagram is going to be more conducive to the exploration than the higher-order logic representation because the relevant information — the interconnections of data — are explicit.

Notice the aspects of design elicited by this argument: the need to look at multiple features or representations of a system and the fact that the clearest representations for a particular property might not be sentential. Designers have traditionally used a number of representations of information during a the design process, and diagrams are commonly among them. The verification community cites these practices as problematic to verification [4] because the method of using multiple representations, some of them diagrammatic, is not rigorous.

Nothing prevents such a methodology from being made rigorous. In fact, making such a methodology logically sound offers a first step towards addressing the proof-versus-reasoning problem. That designers persist in using this methodology despite its lack of formal basis suggests that they are comfortable reasoning in this manner.

Heterogeneous logics [1] are the key to making this methodology rigorous. In a heterogeneous logic, multiple representations of information interact formally via semantics and rules of inference. The first reasoning tool based upon heterogeneous logic was Barwise and Etchemendy’s *Hyperproof* system [2]. *Hyperproof* integrates first order sentential logic with a diagrammatic blocks world, thereby allowing the user to reason with information from both representations simultaneously. *Hyperproof* does not treat its diagrams as mere interface tools; rather, it treats diagrams as valid logical objects on par with sentential formulae. The system contains the standard inference rules of first order predicate logic along with inference rules from sentences to diagrams and vice-versa. Additional details appear in [1] and [2].

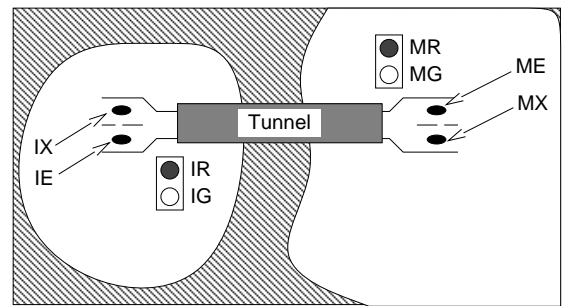
Our logic supports four representations: gate level circuit diagrams, timing diagrams, algorithmic state machines (ASMs) [15], and higher-order sentential logic. The syntaxes of the first three representations are diagrammatic; in each case a classification of the well-formed representations is provided. The logic uses a model of physical hardware devices as a semantic basis. The use of a common semantic basis for the four representations fa-

cilitates the definition of inference rules that bridge multiple representations. Diagrams are not translated into a sentential logic for purposes of inference rule definition; rather, inference rules are defined directly on the diagrammatic representations, in the same manner as they are traditionally defined on formulae.

The diagrammatic portion of heterogeneous hardware logic was first presented in [8]. The current logic is not presented here due to space constraints, but appears in [10] with presentation of inference rules and more detailed examples.

III. THE ISLAND TRAFFIC LIGHT CONTROLLER

Assume that we want to design a controller for the traffic lights at a one lane tunnel connecting the mainland to a small island as pictured below. There is a traffic light at each end of the tunnel; there are also four sensors for detecting the presence of vehicles: one at tunnel entrance on the island side (IE), one at tunnel exit on the island side (IX), one at tunnel entrance on the mainland side (ME), and one at tunnel exit on the mainland side (MX).



In addition, there is a constraint that at most sixteen cars may be on the island at any time. We make the environmental assumptions that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, and that cars do not leave the tunnel entrance without travelling through the tunnel.

Our proposed design consists of the three communicating controllers shown in figure 2: one for the island lights, one for the mainland lights, and one tunnel controller that processes the requests for access issued by the other two controllers. We would like to establish that our solution has at least the following properties:

1. Cars never travel both directions in the tunnel at the same time.
2. Access to the tunnel is not granted until the tunnel is empty.
3. Lights don’t turn green until access is granted.
4. The tunnel is used once granted.

$$\forall t \exists t_1 t_1 > t \wedge \neg IE(t_1) \wedge \forall t_2 t_1 > t_2 \geq t \rightarrow IE(t_2)$$

$$\forall t \exists t_1 t_1 > t \wedge IE(t_1) \wedge \forall t_2 t_1 > t_2 \geq t \rightarrow \neg IE(t_2)$$

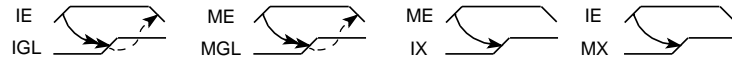


Fig. 1. Heterogeneous representations of the environmental assumptions. In the timing diagram notations, a dashed arrow between two events indicates that the second event must be preceded by the first (safety). A double headed solid arrow indicates that the first event must eventually be followed by the second event (liveness). A single headed solid arrow indicates a combination of these two edge types. Semantically, arrows on events must be satisfied at all times, as opposed to satisfied only once during an execution. These edge notations were originally proposed by Schlör and Damm [16].

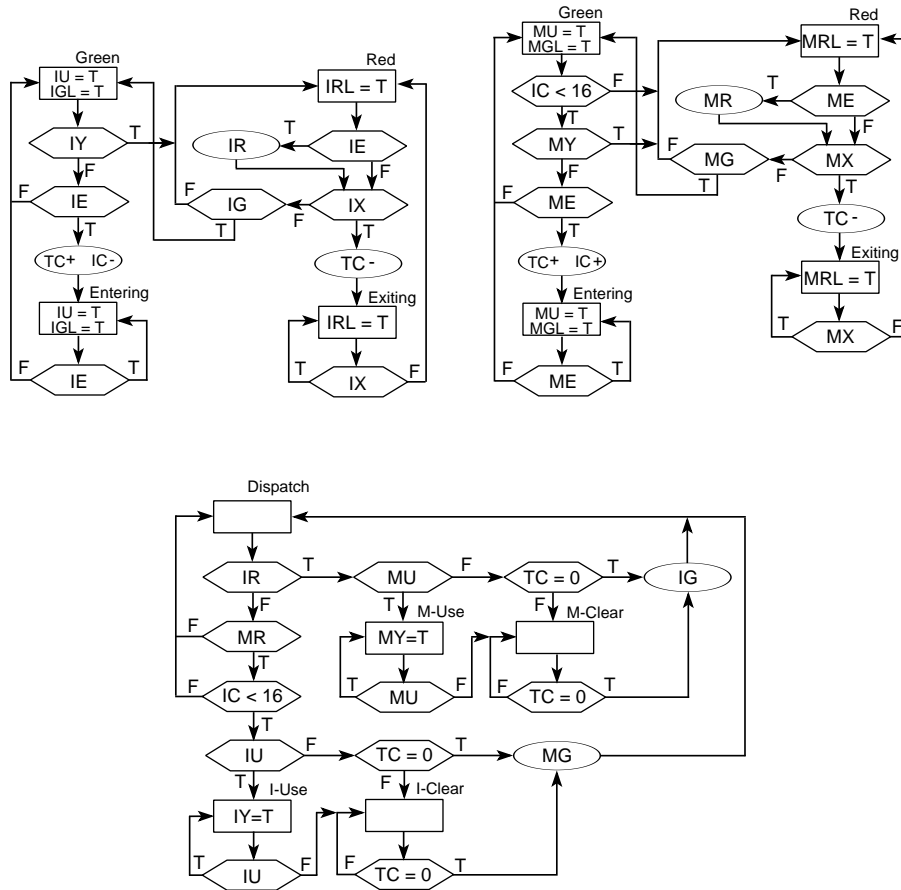


Fig. 2. ASM charts for the Island Traffic Light Controller; the island light controller is on the top left, mainland light controller on the top right, and tunnel controller on the bottom. IGL and IRL are the green and red lights for the island, IU indicates that the island is using the tunnel, IR indicates that the island is requesting the tunnel, IY indicates that the island is being instructed to release control of the tunnel, and IG indicates that the island has been granted control of the tunnel; a similar set of signals has been defined for the mainland. TC is a count of the number of cars presently inside the tunnel and IC is a count of the number of cars presently on the island.

5. Requests for the tunnel are eventually granted.
6. Once a car arrives at an entrance, the light at that entrance eventually turns green.
7. All commands to yield the tunnel are acknowledged by the island and mainland controllers.
8. There are never more than 16 cars on the island.
9. Counters are only changed once per car.
10. Counters behave properly if signalled to increment and decrement simultaneously.

Some of these properties, such as 2 – 7, are natural candidates for finite-state verification techniques. Others, such as 10, are easier to reason about once an implementation of the system is designed; the logic supports reasoning at each of these levels. As an example, we could reason about the condition that all yields issued by the tunnel controller should eventually be acknowledged. This condition can be expressed for the island controller using the timing diagram



in which the double-headed arrow requires that signal IU (“island using the tunnel”) falls concurrently or after signal IY (“island told to yield”) rises.

Given the representation of the environmental conditions provided in figure 1, we can reason about whether or not the design meets the property.¹ Assume we are in a state in which the island is asked to yield the tunnel (indicated by IY being true). Looking at the diagrams in figure 2, observe that the tunnel controller must be in state $I-Use$. Two transitions are possible from this state, depending upon the value of the signal IU . If IU is false, the desired condition is satisfied, so we need only consider the case when IU is true. We must prove that IU eventually becomes false. Note that IY will remain true until IU become false. If IU is true, the island controller must be in state *green* or state *entering*. From state *green* with IY true, the island controller transitions to a state (*red*) in which IU is false, thus satisfying the condition. From state *entering*, we see that the condition will be satisfied if we can prove that we eventually reach state *green*. We reach *green* if signal IE is guaranteed to become false, which is given as one of our environmental conditions, thereby completing our reasoning process.

This example demonstrates that state machines are good representations for reasoning about control behavior; the acceptance of model-checking and language containment verification techniques has demonstrated that

¹Due to space constraints, we can not provide the formal proof and the diagrams used as its supporting steps; as a result, we present the informal reasoning here and refer the reader to [10] for the formal proof.

they are also good representations for certain styles of proof. Although the above reasoning process was deductive in nature, we do not mean to imply that automatic verification techniques should not be used when possible. At present, there is no model-checking algorithm defined for this logic, but given the correlations between diagrammatic and sentential representations of state machines, adding one should be straightforward.

Assuming we are satisfied with the state-level design, the next step becomes designing an implementation of the system. This too can be done using the inference rules of the logic. There are several algorithms for converting a state machine into physical hardware [15]; for this example, we will take a state-encoded approach to the design. The circuit diagrams provided in figure 3 for the island light controller can be shown to be logical consequences of the associated state machine in figure 2.

The diagrams in figure 3 suggest how the logic can be used for design. Assuming we derive the top diagram under an inference rule for state-encoded implementations of state machines, there are a number of optimizations we could make to minimize the number of gates in the circuit. The bottom diagram reflects one possible minimized circuit obtained from the original by means of circuit diagram inference rules. The soundness of the rules is sufficient to assure us that the two circuits are behaviorally equivalent; a proof of completeness of the rules — which allows the rules to transform any two behaviorally equivalent circuit diagrams into one another — is in progress based on a canonical form for circuit diagrams [9].

Given implementations of the three controllers, all that remains is to design the components necessary to interface the three implementations. While some of the interface consists only of wires, additional logic is required to integrate the counters and the needed comparator into the final design. This brings us back to the issue of verification, as we would like to formally establish the correctness of the interface logic.

As an example, consider the interface logic required for the counter **TC** that records how many cars are currently in the tunnel. Assume we have chosen to use a LS191 up-down counter [5] in our implementation. This counter has an enable signal and a single signal for indicating whether the counter should count up or count down. When the enable signal is low, a low voltage on the up/down line causes counting upwards and a high voltage on the up/down line causes counting downwards; no counting occurs when the enable line voltage is high. Once we interface the controller implementations with the counter, we must verify that our interface logic routes signals properly to the LS191. Assume that we used the following interface logic, where $I-TCIncr$ is the signal $TC+$ from the island light controller and the remaining signals are analogously defined.

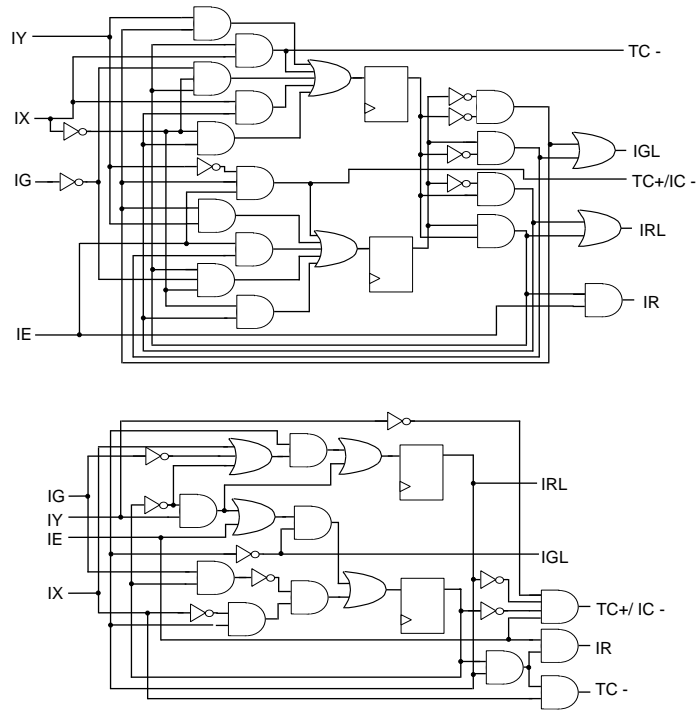
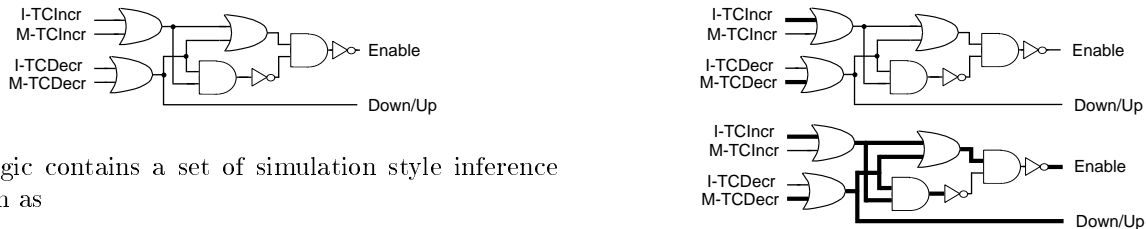
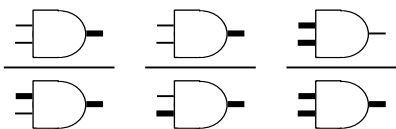


Fig. 3. Circuit diagrams for the island light controller. The top diagram is naïvely produced using a state-encoding of green=00, entering=01, exiting=10, and red=11. The simplified yet behaviorally equivalent bottom diagram is derived from the first using the boolean algebra based circuit diagram inference rules.



The logic contains a set of simulation style inference rules such as



in which thick lines indicate high voltage and thin lines indicate unknown voltage. We will use these rules to verify that our interface logic behaves as desired. As an example proof, consider the case when the island controller issues a tunnel counter increment and the mainland controller issues a tunnel counter decrement.² In this case the counter should hold its current value. The following proof consists of two diagrams. In the first, we assume that both I-TCIncr and M-TCDecr are asserted simultaneously. The second diagram shows the resulting asserted signals once the simulation rules are applied to the first diagram.

IV. CONCLUSIONS

Despite the simplicity of the examples given here, it should be clear that diagrams can play a role in facilitating hardware reasoning; furthermore, diagrams can be used at multiple stages of design, making them good representations for integrating design and verification methodologies. A tool based on such a heterogeneous logic would be a strong step towards bringing usable verification techniques to designers. This logic supports reasoning because it mimics how people use representations outside of the realm of formal methods.

The use of diagrams in hardware reasoning is not a new idea. Many systems have used diagrams as an interface tool, translating them into known sentential logics for purposes of inference and proof [6] [7] [18]; considerable such work has been done using timing diagrams [12] [16]. The integration of verification methodologies within a single system is also not new [13] [17]. What is unique

²This combination is possible in our proposed state machines.

to our approach is the notion of a heterogeneous system supporting diagrams as logical entities on even par with traditional sentential representations; we believe this is a more natural reflection of how diagrammatic reasoning is done in practice.

Despite the potential advantages suggested by our approach, the current state of the research suffers from certain drawbacks when contrasted with other design and verification tools. We currently lack the automation of a model checking system, and we lack the abstract type definitions and flexibility of a more generic theorem proving system. We lack the automatic layout facilities of synthesis and some of the interface required to use the logic as a pure simulation system. However, we believe that these issues generally reflect the early stage of the research rather than any inherent drawbacks to the approach. For example, our ASM charts differ from the SMV specification language only at a syntactic level; Schlör and Damm [16] have shown that their timing diagrams can be translated into PTL. This suggests that portions of the logic could be automated through model-checking. The integrated approach we propose here in turn offers an advantage over straightforward model-checking; logics such as CTL are known to suffer from limitations in expressibility. Once we have a classification as to which of our timing diagrams, for example, can be model-checked, we offer a user the chance to model-check some diagrams and interactively verify others, without the loss in expressive power associated with working within decidable languages.

There is no implementation of a tool based on the logic at this time. While we would like to implement such a tool, present efforts are directed at further developing the theory of heterogeneous hardware logics.

ACKNOWLEDGEMENTS

The authors thank Jon Barwise and Gerry Allwein for many useful discussions on the ideas in this paper. Zijian Zhou found errors in the original diagrams of figure 3.

REFERENCES

[1] Jon Barwise. Heterogeneous reasoning. In G. Mineau, B. Mouline, and J. Sowa, editors, *Conceptual Graphs and Knowledge Representation*. Springer-Verlag, 1993.

[2] Jon Barwise and John Etchemendy. Hyperproof, CSLI Lecture Notes, University of Chicago Press. To appear, 1994.

[3] Jon Barwise and John Etchemendy. Logic, proof, and reasoning. In Alan Makinowski, editor, *Companion to Logic*. Blackwell, To appear.

[4] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[5] National Semiconductor Corporation. LS/S/TTL logic databook, 1987.

[6] L.K. Dillon, G. Kuty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna. A graphical interval logic for specifying concurrent systems. Technical report, UCSB, 1993.

[7] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design — interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *Formal VLSI Specification and Synthesis: VLSI Design-Methods-I*. North-Holland, 1990.

[8] Kathi Fisler. A logical formalization of hardware design diagrams. Technical Report TR416, Indiana University, September 1994.

[9] Kathi Fisler. A canonical form for circuit diagrams. Technical Report TR432, Indiana University, May 1995.

[10] Kathi Fisler. Exploiting the potential of diagrams in guiding hardware reasoning. In Gerard Allwein and Jon Barwise, editors, *Logical Reasoning with Diagrams*. Oxford University Press, To appear, 1995.

[11] M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North Holland, 1986.

[12] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, and A. Silburt. Integrating behavior and timing in executable specifications. In *CHDL*, pages 385–402, April 1993.

[13] Zohar Manna et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.

[14] NASA Langley Formal Methods Workshop, May 1995. Panel Sessions and Discussions.

[15] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design*. Prentice-Hall, 2nd edition, 1987.

[16] Rainer Schlör and Werner Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proc. European Conf. on Design and Automation*, Paris, February 1993.

[17] SRI International. The PVS2 Verification System.

[18] Mandayam Srivas and Mark Bickford. SPECTOOL: A computer-aided verification tool for hardware designs, vol I. Technical Report RL-TR-91-339, Rome Laboratory, Griffiss Air Force Base, December 1991.