MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 967 May, 1987

A Multiple Representation Approach to Understanding the Time Behavior of Digital Circuits

Robert J. Hall*, Richard H. Lathrop*, Robert S. Kirk**

* M.I.T. Artificial Intelligence Laboratory, Cambridge, MA 02139
** Gould Semiconductor Division, Santa Clara, CA 95051

ABSTRACT. We put forth a multiple representation approach to deriving the behavioral model of a digital circuit automatically from its structure and the behavioral simulation models of its components. One representation supports temporal reasoning for composition and simplification, another supports simulation, and a third helps to partition the translation problem. A working prototype, FUNSTRUX, is described.

This paper will appear in *Proceedings of the Sixth National Conference on Artificial Intelligence* (AAAI-87), Seattle WA.

Acknowledgments. This paper was prepared jointly at the Gould Semiconductors CAD Research Laboratory and at the MIT Artificial Intelligence Laboratory. Support for the MIT Artificial Intelligence Laboratory's research is provided in part by the Office of Naval Research under contract N00014-85-K-0124. Personal support for the first author was furnished by an NSF Graduate Fellowship. Personal support for the second author was furnished by an IBM Graduate Fellowship, and during the early stages of this research by an NSF Graduate Fellowship.

1 Introduction

The function (time behavior) of a system is determined by the functions of its parts, together with their structural connections. Unfortunately, understanding the function of the whole by understanding the parts is difficult and poorly understood. The domain of digital circuits is convenient for investigating this problem because many of its objects already have well-defined, machine-readable behavior descriptions. These reside in the simulation model libraries used by the design community.

We have developed a multiple representation approach to *automatically deriving models of overall device behavior* from the interconnection structure of its components, together with their behavior models. Because one of our behavior representations is the executable simulation model, our system can transform executable program code describing the components' behavior into executable program code describing the behavior of the device as a whole.

The naive solution, invisibly simulating the components, has no value. Instead, we transform each component's behavior to a representation that highlights value dependencies over time. These are propagated by substitution according to the circuit structure and simplified to produce an overall behavior description.

We exploit different representations to facilitate different reasoning tasks:

- The *temporal equation* representation facilitates reasoning about dependencies between values at times.
- The *code* representation is the executable simulator model for the circuit.
- The *abstract event* representation helps partition the translations between code and equations.

A working prototype, FUNSTRUX, accepts as input the components' interconnection and simulator models, and produces an executable simulator model for the entire circuit. The input components' descriptions (and the final output) can also be in either of the equivalent abstract event or temporal equation representations.

FUNSTRUX has been tested successfully on the SCORE [1] standard cell library generator system, and has successfully generated a behavioral

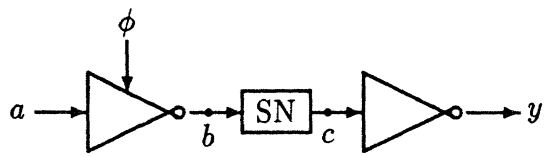

Figure 1: A dynamic storage cell. Its three components (l-r) are a clocked inverter with delay 1.7 nsec, a storage node, and an inverter with delay 0.5 nsec.

model for one bit-slice of an AMD 2901 [3] (an arithmetic-logic unit with memory and control circuitry, having 370 gate-level components and 365 interconnecting buses). Generating the functional model for this large circuit required 7 hours of real time on a Symbolics Lisp Machine. The resulting module uses less than a third as much time to simulate and schedules an order of magnitude fewer events than the full circuit at component level [12].

2 An Example

Behavioral simulators are applications of well-known event-driven simulation techniques to digital circuits. Each circuit component has an algorithmic description which dictates how it propagates value-changes (events).

Consider the circuit shown in Figure 1. For the purposes of this example, we choose a simple set of three logic levels (values): 1, 0, and $*$ ("no value," e.g. the value of a tri-stated device). The inverter unconditionally puts out the negation of its input at a delay of 0.5. (Throughout, times will be in units of nanoseconds.) If the clocked inverter's ϕ input is 1, then its output becomes the logical negation of its a input 1.7 time units into the future. Otherwise, its output will be $*$ 1.7 into the future. The storage node holds the most recent non-$*$ value. (This example has been simplified from [12] and reflects minor improvements in FUNSTRUX' code since that paper.)

FUNSTRUX produces the following code from this example. (Only the code for the y output is shown here.)

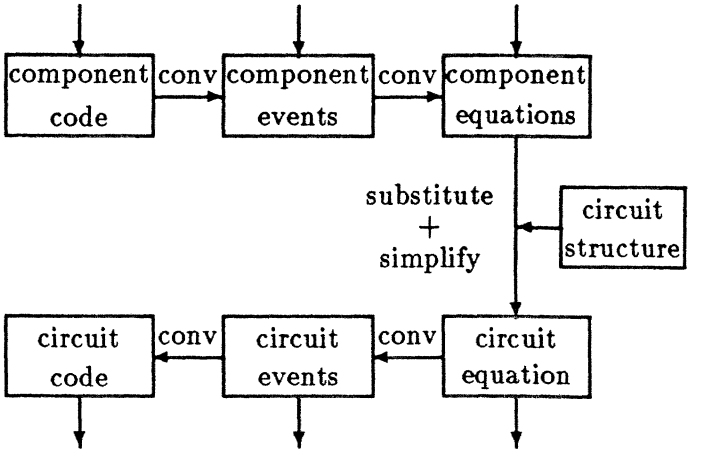

Figure 2: The FUNSTRUX system has two sorts of module. Representation conversion modules change one view of a circuit into another. Substitute/simplify modules perform various sorts of simplification.

```
(defun dynamic-storage-cell-fcn (self a y phi)
   (depends-on '(a phi) '()
      (if (logone (read-bus phi '(bit)))
          (put-my-state self '(a-state)
             (read-bus a '(bit)) 0.0)))
   (depends-on '() '(a-state)
      (drive-bus y self '(bit)
         (get-my-state self '(a-state)) 2.2)))
```

This says that "when either a or ϕ changes, if ϕ is 1, schedule an event at the same time to change a_{state} to the current value of a. Whenever a_{state} changes, schedule an event 2.2 later to drive y to the value of a_{state}." This description omits the double negation and the details of the storage node changing value.

3 The Representations and Conversions

The FUNSTRUX system's multiple representation scheme (see [16]) is shown in Figure 2. It runs as one component of the STAR design system

[11] and continues an investigation into function, structure, and their relationships. The STAR system integrates behavioral simulation [13], netlist manipulation [14], and parameterized cell generators [1]. STAR's LISP-based behavioral simulator, SIMMER, is our target.

SIMULATOR CODE REPRESENTATION. The code representation is described in [13]. The SIMMER model for the storage node is

```
(defun storage-node-fcn (self b c)
   "b input, c output"
   (let* ((b-value (read-bus b '(bit))))
      (if (≠? * b-value)
          (put-my-state self '(bit) b-value))
      (drive-bus c self '(bit)
         (get-my-state self '(bit)))))
```

ABSTRACT EVENTS REPRESENTATION. Different event-driven simulators have differing semantics and coding conventions; however, they share a common event-driven core. The abstract event representation partitions the problem of translating from code to equations (and back) into (1) a simulator-dependent segment which abstracts from the particular semantics of the simulator into abstract events, and (2) a simulator-independent segment which translates from the event representation to the equation representation. This will make it easier to re-target the system for different behavior descriptions.

An abstract event description consists of the event condition predicate, the variable to be scheduled for change, the relative delay, and the symbolic expression for the new value. The event condition predicate consists of two parts, the enablement predicate (ep) and the changing-variable (cv) list. The event occurs if *either* the enablement predicate becomes true *or* one of the changing-variables changes value while the enablement predicate is true. [15] used a notation similar to our enablement predicate (ep).

The abstract event representation of the storage node, produced from the code above, is

```
(Event-cell inputs:(b)
   event:((cv (b) ep (≠? * b))
          (schedule b_state at 0 equal-to b))
   event:((cv(b_state) ep T)
          (schedule c at 0 equal-to b_state)))
```

This represents a cell with one input, b. $b_{\mathbf{state}}$ represents the memory of the storage node, and c is the output. The first **event** clause indicates that when either the value of b changes and the predicate $[\neq? * b]$ is true, or when the predicate changes from false to true, an event is scheduled at the same time which sets $b_{\mathbf{state}}$ to the value of b. (Note that in the example b is not * only when the clocked inverter is driving.) The second **event** clause schedules a change of c, but since the enablement predicate is T it happens on any change of $b_{\mathbf{state}}$.

TEMPORAL EQUATIONS REPRESENTATION. One key insight in our approach is that one needs a behavior representation with *locality of reference* among circuit values at given times. This means that the time points relevant to the computation of a value must be explicit, and that circuit value dependencies should be explicit and local to uses of the circuit values. For example, applicative programs have locality of reference, while programs which set and use global variables at widely separated places do not. [5] has emphasized the importance of locality for reasoning about the behavior of circuits.

Timelines are mappings of the real numbers into values. We view circuits as mappings between timelines. [10] used a timeline notion, but their time domain was discrete.

A circuit output at time t is expressed in a temporal equation as an applicative function of its inputs at the same or earlier times. We use two primitive *time operators*, $-$ and $\Leftarrow$ for expressing values at earlier times. The second argument to $-$ must be a non-negative constant. $\Leftarrow$ allows reference to "the most recent time a predicate was true." Thus, $\{\overset{u}{\Leftarrow}\ [Pu]\ t\}$ refers to the most recent time, u, prior (or equal) to t, such that the predicate P was satisfied at u. This is similar to the left-arrow operator of [17]; however, they also work with a discrete time domain. [2] proposed a similar applicative formalism for representing and reasoning about circuits, but did not automate the reasoning and did not relate the representation to simulation.

We represent computation on values as functional application in a side-effects-free LISP-like format. (We have converted prefix to infix notation here for readability.) The temporal equations for the example's components are

Clocked Inverter: $b(t) = (\text{if } (=?\ 1\ \ \phi(t-1.7)) \quad ;\text{test}$
$\neg\ a(t-1.7) \quad ;\text{then}$
$*) \quad ;\text{else}$

Storage Node: $c(t) = b(\{\overset{u}{\Leftarrow}\ [\neq?\ *\ b(u)]\ \ t\})$

Inverter: $y(t) = \neg\ c(t-0.5)$

REPRESENTATION CONVERSIONS. The representation conversion algorithms are treated in more detail in [12]. Here are the key ideas.

- *Code → Abstract Events.* The code is symbolically executed to associate each symbol with a formula which computes its value under the appropriate conditions. Unknown forms are treated as "black-box" functions according to the semantics of pure LISP. Forms which perform side-effects (e.g., reading or setting a global variable) are not modeled correctly.

- *Abstract Events → Equations.* A value doesn't change between events, so the value at time t is the value of whichever event expression occurred most recently. By constructing a predicate which indicates when a variable's value last changed, we are able to reason about the last time an event would have triggered.

- *Equations → Abstract Events.* "State objects" may be created to conditionally delay the values of the inputs.

- *Abstract Events → Code.* Each variable can be resolved into either an I/O port or a state object, using the circuit structure. Language constructs can then be generated which produce the effect of each event.

SEMANTIC CONNECTIONS BETWEEN THE REPRESENTATIONS. A multiple representation scheme must at some point answer the question of semantic equivalence of different representations. We view the equation representation as a notation for a *denotational semantics* for the circuit structure. A circuit, together with inputs and initialization, denotes the unique solution to the simultaneous equations. The abstract event representation (and the simulator code) are endowed with event-based *operational semantics.* [8] shows that our equation representation is equivalent to an essentially similar abstract events representation as long as zero-delay loops

are disallowed. Proving equivalence of the code and abstract events representations is an open problem.

4 Locality, Simplification, and Temporal Reasoning

COMPOSITION. Composing the behaviors of the components in equation representation amounts to algebraic substitution of equations. This process maintains locality of reference: when a reference to $b(t)$ is expanded by replacing it with b's definition, the variables on which b depends appear explicitly everywhere b was used.

Here is the fully substituted example:

$$
\begin{array}{l}
y(t) = \neg\ (\text{if } (=?\ 1\ \phi(\{\overset{u}{\Leftarrow}\ [\neq?\ *\ (\text{if } (=?\ 1\ \phi(u{-}1.7)) & \text{;test} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \neg\ a(u{-}1.7) & \text{;then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad *)] & \text{;else} \\
\qquad\qquad\qquad\qquad\qquad\qquad t{-}0.5\} & \\
\qquad\qquad\qquad\qquad\qquad -1.7)) & \\
\qquad\qquad \neg\ a(\{\overset{v}{\Leftarrow}\ [\neq?\ *\ (\text{if } (=?\ 1\ \phi(v{-}1.7)) & \text{;test} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \neg\ a(v{-}1.7) & \text{;then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad *)] & \text{;else} \\
\qquad\qquad\qquad\qquad\qquad\qquad t{-}0.5\} & \\
\qquad\qquad\qquad\qquad\qquad -1.7) & \\
\qquad\qquad *)\ \text{; final else clause} &
\end{array}
$$

On circuits of even moderate complexity, the combinatorial explosion is much worse; hence, simplification is needed. FUNSTRUX interleaves substitution and simplification to reduce intermediate expression size.

PATTERN-ACTION SIMPLIFIERS. Our system's syntactically local representation supports simple pattern-action expression transformations, similar to those used by [4] for program optimization. The rules in FUNSTRUX are tailored to *simplification.* They form a *terminating rule set,* so the system applies them until no more are applicable. Experience has shown that we do not need to search different application orders.

One simplifier applicable to the equation above is

$$(\neq?\ *\ (\text{if } \textit{predicate value}\ *))$$

$$\xrightarrow{\text{simplifies to}}$$

$$(\text{AND } predicate\ (\not=?\ *\ value))$$

We have implemented a symbolic simplifier which uses approximately 50-75 rules of this type. It typically reduces the size of the expressions by about 90%. Syntactic locality is crucial to the efficiency of this technique, as hunting all over a non-local representation would slow down the pattern matchers. Furthermore, the action parts would be less efficient, as relatively major surgery would be required.

Simplifiers free of time operators could also be applied to the abstract event and/or code representations. However, the same can *not* be said for pattern-action simplification which involves time relationships.

REASONING ABOUT TIME RELATIONSHIPS. Locality of reference among time relationships of variables is also important. First, there are several useful time-based pattern-action simplifiers. For example, this is applicable to the equation above:

$$\{\overset{u}{\Leftarrow}\ [predicate\ (u-\gamma)]\ \ t\}-\gamma$$

$$\xrightarrow{\text{simplifies to}}$$

$$\{\overset{u}{\Leftarrow}\ \ [predicate(u)]\ \ (t-\gamma)\}$$

Subtracting γ from the most recent time $u \leq t$ such that *predicate* is true at $u-\gamma$, is the same as the most recent time $u \leq t-\gamma$ that *predicate* is true.

Applying all of the system's pattern-action simplifiers to the example, we get

$$\begin{array}{lll} y(t) = (\text{if } (=?\ 1\ \phi\ (\{\overset{u}{\Leftarrow}\ \ [=?\ 1\ \phi(u)]\ \ (t-2.2)\})) & ; \text{ test} \\ \qquad a(\{\overset{u}{\Leftarrow}\ \ [=?\ 1\ \phi(u)]\ \ (t-2.2)\}) & ; \text{ then} \\ \qquad \neg\ *) & ; \text{ else} \end{array}$$

Another crucial feature of the locality property is that it exposes exactly the set (relative to t) of time points which are relevant to the output value. These are just the ones explicitly mentioned in the equation (t, $(t-2.2)$, and $\{\overset{u}{\Leftarrow}\ \ [=?\ 1\ \phi(u)]\ \ (t-2.2)\}$), plus $-\infty$. *Note that we have reduced the problem from reasoning about $\forall t$ and $\exists t$ to the much easier problem of propositional reasoning about a finite number of time points.* It is an open question whether our system will need to reason about any other times than these for

simplification; however, we have not yet come across any examples which indicate that it will.

We have implemented a propositional reasoner to support reasoning about the truth of predicates at time points. This is needed in order to incorporate some types of *background knowledge*, such as "if a value is known to be 1 at a time, it is not also 0 at that time;" to handle certain kinds of *simplifying assumptions* [7]; and to support simplifications based on logical conditions implied by the context of an expression, for example, predicates which are true due to nesting within a conditional.

In the example, $(=?\ 1\ \phi(\{\overset{u}{\Leftarrow}\ [=?\ 1\ \phi(u)]\ \ (t{-}2.2)\}))$ can not be simplified to TRUE, because it could be that ϕ has never been 1 prior to $t - 2.2$. Frequently, however, we wish to consider only the normal-case behavior of the circuit, in which ϕ will have been 1 prior to $t - 2.2$ for any t under consideration. We can communicate this simplifying assumption to our system by the axiom

$$\forall(t > -\infty).\{\overset{u}{\Leftarrow}\ [=?\ 1\ \phi(u)]\ \ t\} > -\infty$$

The system reduces this axiom from a universal quantification to a proposition for each time point in the equation, and concludes, through propositional reasoning, that the predicate is true. With the other simplifiers, this produces

$$y(t) = a(\{\overset{u}{\Leftarrow}\ [=?\ 1\ \phi(u)]\ \ (t{-}2.2)\})$$

as the simplified equation for the example. Converting this to abstract events, the system produces

```
(Event-cell inputs:(a φ)
   event:((cv(a φ) ep(=? 1 φ))
          (schedule a_state at 0.0 equal-to a))
   event:((cv(a_state) ep T)
          (schedule y at 2.2 equal-to a_state))
```

The system then converts this to the code in Section 2.

5 Conclusions and Future Work

We have explained our multiple representation approach to understanding the time behavior of digital circuits. To our knowledge, this is the first system to accept program code for the functional models of the circuit components,

together with their structural connections, and produce the program code for the circuit model as a whole.

- The *equation-based* representation makes easier several forms of reasoning about the time behavior of digital circuits. *Locality of reference* is the crucial property of this representation.
- The *code-based* representation makes simulation efficient.
- The *abstract event-based* representation partitions the translation problem between code and equations.
- Unknown forms in the program code are treated as black-boxes according to the semantics of pure LISP.
- The system's local representations support efficient pattern-action simplification.
- The finitely-many relevant time points are made explicit, allowing propositional reasoning for time-based simplification.

This work is preliminary and represents only a first step toward our goal. Currently, FUNSTRUX is limited in the class of circuits to which it can be applied. The restrictions are (1) busses can connect only to blocks (not to each other), (2) busses change state only when driven by a block, and (3) zero-delay loops are disallowed (in particular, this disallows zero-delay bidirectional elements).

Here are a few issues for further research.

- There are several interesting reasoning tasks in the realm of digital circuits to which we hope to extend our representation scheme. Some examples: *design optimization* [19], *troubleshooting* [6], *testing* [18], and *learning about design* [9]. Each of these tasks requires its own representations.
- There are several ways the system could be improved: it currently does not find closed forms for the recursion equations which result from feedback; it could allocate state objects for simulation better than it currently does; it could recognize low-level implementations of higher level functions, such as integer +.

- The code which is output by FUNSTRUX is not organized for readability.
- It may be useful to incorporate work on pattern-action simplification of VLSI structure [14].
- What constraints must be met by a particular simulator in order that the abstract event representation be able to capture an equivalent meaning?

Acknowledgments

The authors would like to acknowledge helpful discussions with Mark Alexander, Walter Hamscher, Chuck Rich, Ron Rivest, Brian Williams, and Patrick Winston.

References

[1] Mark Alexander. A spatial reasoning approach to cell layout generation. In *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, IEEE, May 1986.

[2] P. Amblard, P. Caspi, and N. Halbwachs. Describing and reasoning about circuits behavior by means of time functions. In *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and their Applications*, IFIP, 1985.

[3] AMD. *Bipolar Microprocessor Logic and Interface, AM2900 Family Databook.* Advanced Micro Devices, 1985.

[4] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16, 1981.

[5] Randall Davis. Diagnosis via causal reasoning: paths of interaction and the locality principle. In *Proceedings of the Third National Conference on Artificial Intelligence*, AAAI, 1983.

[6] Randall Davis and Howard Shrobe. Representing structure and behavior of digital hardware. *Computer*, 16(10), October 1983.

[7] Yishai Feldman and Charles Rich. Reasoning with simplifying assumptions: a methodology and example. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, Morgan Kaufmann, 1986.

[8] Robert J. Hall. A fully abstract denotational semantics for event-based simulation. In *Proceedings of the Fifteenth Conference on Applied Simulation and Modelling*, IASTED, 1987.

[9] Robert Joseph Hall. Learning by failing to explain. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, AAAI, 1986.

[10] Van E. Kelly and Louis Steinberg. The CRITTER system: analyzing digital circuits by propagating behaviors and specifications. In *Proceedings of the Second National Conference on Artificial Intelligence (AAAI-82)*, AAAI, 1982.

[11] Robert S. Kirk, Robert J. Hall, and Richard H. Lathrop. SCORE cell development environment. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC-87)*, IEEE, 1987.

[12] Richard H. Lathrop, Robert J. Hall, and Robert S. Kirk. Functional abstraction from structure in VLSI simulation models. In *Proceedings of the 24th Design Automation Conference*, IEEE, 1987.

[13] Richard H. Lathrop and Robert S. Kirk. An extensible object-oriented mixed-mode functional simulation system. In *Proceedings of the 22nd Design Automation Conference*, IEEE, 1985.

[14] Richard H. Lathrop and Robert S. Kirk. A system which uses examples to learn VLSI structure manipulation. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, AAAI, 1986.

[15] P. Meinen. Formal semantic description of register transfer language elements and mechanized simulator construction. In *Proceedings of the 4th IEEE International Symposium on Computer Hardware Description Languages*, IEEE, 1979.

[16] Charles Rich. The layered architecture of a system for reasoning about programs. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985.

[17] R. L. Schwartz, P. M. Melliar-Smith, F.H. Vogt, and D.A. Plaisted. *An Interval Logic for Higher-Level Temporal Reasoning.* Contractors Report: Contract Number NAS1-17067, National Aeronautics And Space Administration, 1983.

[18] Mark H. Shirley. Generating tests by exploiting designed behavior. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, Morgan Kaufmann, 1986.

[19] Louis I. Steinberg and Tom M. Mitchell. A knowledge based approach to VLSI CAD: the REDESIGN system. In *Proceedings of the 21st Design Automation Conference*, IEEE, 1984.