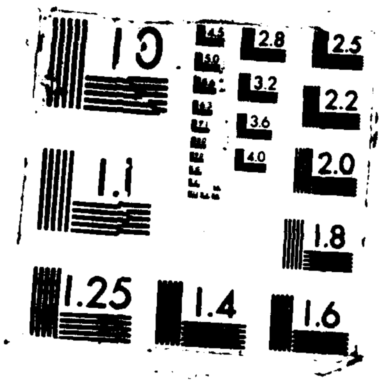
AD-A183 647 FUNCTIONAL ABSTRACTION FROM STRUCTURE IN VLSI SIMULATION MODELS(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB R H LATHROP ET AL. MAY 87 AI-M-903 1/1

UNCLASSIFIED F/G 12/5 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-Memo No. 903	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Functional Abstraction From Structure in VLSI Simulation Models		5. TYPE OF REPORT & PERIOD COVERED AI-Memo
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard H. Lathrop, Robert J. Hall, Robert S. Kirk		8. CONTRACT OR GRANT NUMBER(s) ONR contract N00014-85-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May, 1987
		13. NUMBER OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI circuits, integrated circuits, simulation, functional models, behavioral models, circuit simulation, circuit verification, functional simulation, behavioral simulation, function and structure, temporal reasoning, temporal algebra, temporal logic, syntactic simplification, function from structure.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

High-level functional (or behavioral) simulation models are difficult, time-consuming, and expensive to develop. We report on a method for automatically generating the program code for a high-level functional simulation model. The high-level model is produced directly from the program code for the circuit components' functional models and a netlist description of their connectivity. A prototype has been implemented in LISP for the SIMMER functional simulator.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo 903 May, 1987

Functional Abstraction From Structure in VLSI Simulation Models

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

* MIT Artificial Intelligence Laboratory, Cambridge, MA 02139
** Gould Semiconductor Division, Santa Clara, CA 95051

ABSTRACT. High-level functional (or behavioral) simulation models are difficult, time-consuming, and expensive to develop. We report on a method for automatically generating the program code for a high-level functional simulation model. The high-level model is produced directly from the program code for the circuit components' functional models and a netlist description of their connectivity. A prototype has been implemented in LISP for the SIMMER functional simulator.

This paper will appear in *Proc. of the 1987 Design Automation Conference.*

This paper was prepared jointly at the Gould Semiconductor 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 IBM Graduate Fellowship, and during the early stages of this research by an NSF Graduate Fellowship. Personal support for the second author was furnished by an NSF Graduate Fellowship.

INTRODUCTION.

Highly complex behavior in modern VLSI devices is attained by the complex arrangements of simple elements: transistors are organized into inverters, inverters into flip-flops, flip-flops into register banks, and so on up to large circuits such as microprocessors. Verifying the correctness of complex behavior, in the device *as designed,* is one of the greatest challenges facing the industry today. Currently, simulation is the primary method used to verify the correct behavior of designs. Circuit-level, switch-level, logic-level, and functional (or behavioral) level simulators are used for larger and larger circuits. Higher-level simulation functional models are important to reduce simulation times on large circuits. They are also important for comprehensibility: a single high-level functional model is easier to understand than many low-level models and a description of how they are connected.

Though necessary and desirable, functional models are expensive, difficult, and time-consuming to develop [10]. Because of the cost and engineering effort involved, several companies have even begun to produce functional simulation models commercially [4, 17], producing both component-level and board-level models. There is, however, a more insidious aspect than time, difficulty, and expense: that of error in the functional model. The problem of verifying the correctness of the device to its specifications has been *transformed* to the problem of verifying the correctness of the functional model to the device.

We report work on the problem of automatically generating program code for the higher-level functional simulation model of the device, starting with a description of its circuit. This will allow a circuit designer to create a design, abstract it, and simulate it more efficiently at a higher level.

Our approach builds on AI-based ideas of symbolically manipulating multiple representations, which we applied to symbolic representations of function and time. Key ideas include:

- Multiple representations of knowledge about behavior and time: in LISP code, in algebraic equations, and in an event formalism.
- Capturing program code semantics in the translation between representations.
- Symbolic reasoning about behavior and time.
- An algebraic symbol manipulator for temporal equations.
- Simplification of expressions based both on syntactic form and on time dependencies.

A LISP-based prototype (FUNSTRUX) has been implemented to explore these ideas, and produces executable simulator program code comprising the functional model of a circuit. FUNSTRUX is written in LISP and runs on a Symbolics LISP machine. Input to the system is the SIMMER [21] simulator functional model (coded in LISP) for each circuit component type, together with a netlist description of circuit connectivity. Output is a single SIMMER functional model (also coded in LISP) of the circuit as a whole.

Project goals were primarily intended to demonstrate this basic concept:

- Produce a single abstract functional model for a circuit, from its component functional models and its connectivity.
- Accept executable program code as input and produce executable program code as output.
- Real functional abstraction must be done, i.e. the abstract model may not naively re-encode the individual simulations of its components.
- The functional model must be functionally correct, both in signal values and in time behavior, with respect to the simulator behavior of the components' functional models.
- The program code must be "reasonable", and must speed the simulation, but highly optimized code was not a design goal.
- Appropriate state objects and manipulations must be inferred as necessary to maintain functional correctness, but need not be highly optimized.

We have used this prototype to automatically generate functional models for the SCORE [3] standard cell library generator system, as well as a functional model of a 2901 microprocessor 1-bit slice plus control logic [2], from primitive devices such as latches, clocked inverters, etc. This demonstrates the prototype capability to produce both a standard cell functional library and a complex circuit model automatically from design primitives.

BACKGROUND: STAR SYSTEM AND SIMMER.

FUNSTRUX runs as one component of the STAR design system [20], and continues an investigation into function, structure, and their relationship. STAR is an integrated LISP-based design system supporting a wide variety of powerful tools, designed around the Y model proposed by Gajski and Kuhn [11]. The heart of the system is the data representation schema called the Y-Database. The Y Hardware Description Language is embedded in LISP and is similar to DPL, except that it has been extended to model not just physical layout information but also structure (blocks and connectivity) and function. The data representaton is object oriented and uses the Symbolics Flavor system. The STAR system integrates functional simulation (SIMMER[21]), netlist manipulation (CONSTELLATION[22]) and parameterized cell generators (SCORE[3]) through the Y-Database.

SIMMER [21] is a LISP-based functional simulation system which models functionality in the well-known discrete-event-driven framework. Events (a value transition at some port or state object) invoke the circuit objects involved to execute their functional models, possibly spawning new events as these propagate new values. SIMMER was designed primarily as an experimental simulation framework to facilitate explorations into function and structure such as described in this paper.

FUNCTION FROM STRUCTURE.

FUNSTRUX employs three different representations of the same functional behavior, each useful for a different task: code-based, event-based, and equation-based (see figure 1). The program code comprising the low-level functional models is converted to an event-based representation, using the circuit connectivity information. The events are converted to an equation-based temporal algebra, and then merged by performing the indicated substitutions. One global equation is generated for each external port, and one for each internal state variable referenced by the port equations. Simplifications are performed by syntax-based transformations. The resulting global equations are converted back to the event-based representation. State objects are created and value references resolved between state objects and I/O ports. Finally, the event-based representation is converted back to one large functional program for the entire structural network.

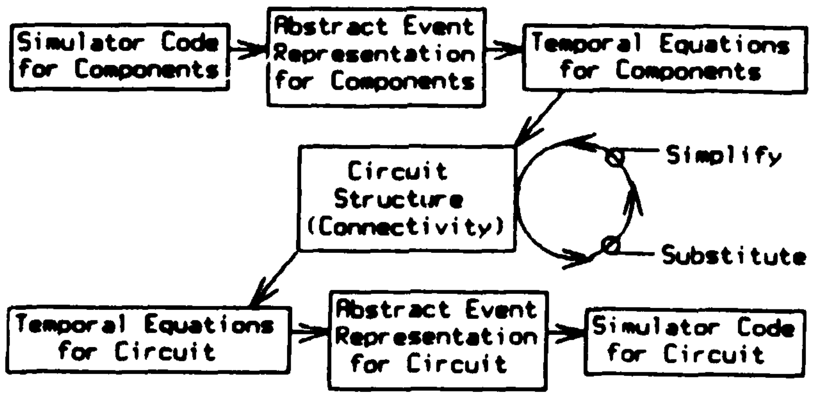

Simulator Code for Components
Abstract Event Representation for Components
Temporal Equations for Components
Circuit Structure (Connectivity)
Simplify
Substitute
Temporal Equations for Circuit
Abstract Event Representation for Circuit
Simulator Code for Circuit

Figure 1. The FUNSTRUX system architecture. Program code for the functional models of the circuit components is converted to equations in a temporal algebra, substituted and simplified according to the circuit structure, and converted back to functional model program code for the whole circuit.

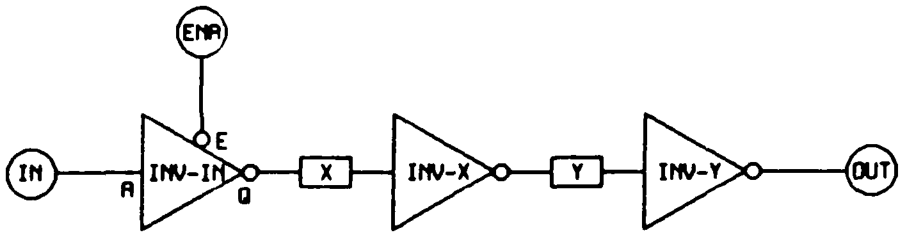


Figure 2. A simple latch made from clocked and unclocked inverters. The example has been contrived to demonstrate the creation of state objects and the collapsing of intermediate structure.

Code-based Representation of Behavior.

The code-based representation (figure 3) is the executable program code making up the SIMMER functional models of the components. (SIMMER code is essentially LISP code, with some built-in functions which read and drive busses, and get and put internal state objects.) Together with a netlist description of component connectivity (generated automatically by the STAR system), the code-based representation is the primary input to FUNSTRUX. It is also the final output, as program code for the whole circuit is produced.

Composing functional models from the program code is difficult. Event schedulings are buried deeply in the code, usually depending on widely-spaced conditional assignments to local variables and other complications. The first transformation is to convert the LISP-based SIMMER code into a description of the events (the conditionalized signal transitions) it implies.

Events are extracted by symbolically executing the programmatic functional model. At each point in the symbolic execution, a binding stack associates each symbol (variable) with a symbolic formula which computes its value at that point from constants, bus values, and state objects. A condition stack maintains the set of overall conditionalizations which must have held for that point of the code to have been reached. This allows determination of what events would have occurred, under what conditions, and how the val-

```
(DEFUN CLOCKED-INVERTER-FCN (SELF Q A E)
  "outputs NOT A to Q if E=1, else tri-states"
  (DRIVE-BUS A SELF '(BIT) (VV :* :Z))
  ;; 'a' gate is always high impedance = <* Z>
  (DRIVE-BUS E SELF '(BIT) (VV :* :Z))
  ;; 'e' gate is always high impedance = <* Z>
  (LET* ((DELAY 1.7)
         ;; worst case E to Q delay
         (A-STATUS (READ-BUS A '(BIT)))
         ;; the 'a' input --- data
         (E-STATUS (READ-BUS E '(BIT)))
         ;; the 'e' input --- enable
         (Q-STATUS (COND ((=? 1 E-STATUS)
                          (VV (NOT A-STATUS)
                              :DRIVE))
                         ((=? 0 E-STATUS)
                          (VV :* :Z))
                         (T (VV :X :DRIVE)))))
    ;; drive 'q' with the result to be output
    (DRIVE-BUS Q SELF '(BIT) Q-STATUS DELAY)))
```

Figure 3. The SIMMER code for the clocked inverter. The LET* establishes variable bindings; the COND selects a value corresponding to the first true test of E-STATUS. If the enable line (E) is logical one, the input (A) is logically negated and output (Q). If E is logical zero, Q is high impedance (:* :Z). Otherwise an X is output.

ues involved would have been derived. Program code semantics are captured based on the class of LISP form:

- Binding forms, such as SETQ and LET*, alter the formula symbolically bound to a symbol on the binding stack.
- Conditional forms, such as COND, IF, UNLESS, and WHEN, push their conditionals onto the global condition stack in an implicit AND.
- SIMMER input forms, such as READ-BUS and GET-MY-STATE, return values.
- SIMMER output forms, such as DRIVE-BUS and PUT-MY-STATE, create

events.

- The SIMMER conditional **DEPENDS-ON** form, which is ignored by FUNSTRUX.
- All other (unknown) forms are treated as "black-box" functions according to the semantics of pure LISP.

This approach allows most standard unknown forms (such as special-purpose functions coded by a designer for a particular purpose) to be handled correctly, though simplification may not be possible. Forms that perform side-effects (e.g., reading or setting a global variable) are not handled correctly.

Following the substitution process described below, the resulting global circuit events are converted back to program code. Events which schedule I/O ports generate **DRIVE-BUS** calls, while events which schedule state objects generate **PUT-MY-STATE** calls. The enablement predicate (:EP, below) is translated into the appropriate conditionalization, and the input variable names (:CV) are used to generate the **DEPENDS-ON** forms. Any "black-box" functions are simply inserted where they appear.

FUNSTRUX is not a simulation language, and so does not attempt to capture the semantics of the hardware directly. Instead, it models the semantics of the underlying simulation language, to produce an abstract functional model which remains faithful to the component-wise simulated behavior of the circuit.

Event-based Representation of Behavior.

The event-based representation (figure 4) highlights signal transitions. It details which values will be output by the functional model at any invocation, conditionalized by which predicates must be true to enable the events. It captures the implied value transitions, specified implicitly by the program code, in an explicit, quasi-canonical, easily-manipulable form.

Each event clause represents a conditionalized signal transition at some port or state object, and so each circuit object's program code potentially

```
((:EV (:CV (INV_IN_E INV_IN_A) :EP T)
      (:SCHEDULE INV_IN_Q :AT 1.7
       :EQUAL-TO
         (COND ((=? 1 INV_IN_E)
                (VV (NOT INV_IN_A) :DRIVE))
               ((AND (=? 0 INV_IN_E)
                     (NOT (=? 1 INV_IN_E)))
                (VV :* :Z))
               ((AND T
                     (NOT (=? 0 INV_IN_E))
                     (NOT (=? 1 INV_IN_E)))
                (VV :X :DRIVE))
               (T NIL))))
 (:EV (:CV NIL :EP T)
      (:SCHEDULE INV_IN_E :AT 0.0
       :EQUAL-TO (VV :* :Z)))
 (:EV (:CV NIL :EP T)
      (:SCHEDULE INV_IN_A :AT 0.0
       :EQUAL-TO (VV :* :Z))))
```

Figure 4. The events generated for the clocked inverter. The meaning of each field is described in the text.

generates several event clauses. The idea is that when either the enablement predicate *becomes* true or the input variables' values change while the enablement predicate *is* true, the value assigned to a port or state object is re-calculated and re-assigned. Accordingly, each event clause defines (where a variable represents a port or state object value):

- The enablement predicate (:EP) on whose leading edge the event occurs;
- A list of the input variable names (:CV) which can trigger the event when their values change;
- Some variable name (:SCHEDULE) which is the recipient;
- The non-negative delay (:AT) associated with the event;
- A symbolic formula (:EQUAL-TO) for the value.

Equation-based Representation of Behavior.

The equation representation (figure 5) facilitates composition, simplification, and reasoning about the code. Behavior is described in an algebraic notation, using temporal operators, which states relationships between port values. This is useful because algebraic equations can be easily composed, and the time behavior of components easily chained together and reasoned about.

$$
\begin{aligned}
inv\_in\_Q(t) = (cond\ &((=?\ 1\ \ inv\_in\_E(t-1.7))\\
&\quad(vv\ (not\ \ inv\_in\_A(t-1.7))\\
&\qquad :drive))\\
&((=?\ 0\ \ inv\_in\_E(t-1.7))\\
&\quad(vv\ \ :*\ \ :z))\\
&(T\quad(vv\ \ :x\ \ :drive))))\\
inv\_in\_E(t) = (vv\ \ :*\ \ :z))&\\
inv\_in\_A(t) = (vv\ \ :*\ \ :z))&
\end{aligned}
$$

Figure 5. The equations generated for the clocked inverter. Each port is shown as a time-dependent function of the time-varying values of the other ports. Note the use of forms like COND, which are well-defined mathematical functions as well as LISP language forms.

We use a continuous model of time, and so there is no "smallest" timestep as in discrete time models. The specification of the value of a variable (a port or state object) as a function of time is a *timeline*. We view devices as mappings on timelines [1, 19]. The $\Leftarrow$ ("left arrow") operator allows reference to the most recent time that a condition (predicate) was true [1, 23, 26]. Thus

$$\{\Leftarrow (u)\ \ (=?\ \ 1\ \ \ Y(u))\ \ (t-\delta)\}$$

returns the most recent time, u (a formal variable), that $Y(u)$ was equal to logical one, but prior or equal to the time $(t-\delta)$.

We represent function by LISP-like algebraic expressions involving no side-effects. Thus, a zero-delay inverting D-Flip-Flop could be represented as

$$Q(t) \;=\; (NOT\;\; D(\{\Leftarrow (u)\;\; (=?\;\; 1\;\;\; CLOCK(u))\;\; (t)\})).$$

This says that the value of the output $Q(t)$ is the logical negation of the input $D(u)$, where u is the most recent time that $CLOCK(u)$ was logical one, but prior or equal to the time t.

Though this facilitates substitution and composition, much needless redundancy and unused code is introduced by these operations. This results in a combinatorial explosion in expression size unless simplification is performed at each step. We have a database of simple, syntactic simplification rules which are very fast to use. For example, $(NOT\;\; (NOT\;\; X))$ simplifies to X.

We also have a reasoner which is capable of time-based logical reasoning for simplification. For example,

$$(=?\;\; 0\;\; X(\{\Leftarrow (u)\;\; (=?\;\; 1\;\; X(u))\;\; (t)\}))$$

simplifies to FALSE: because X was *not* logical zero at the most recent time that it was logical one.

Equations may be converted to and from event representation. The key insight for converting events to equations is that a value does not change between events, so the value at time t is just the value of whichever event expression has 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. The key insight for converting equations to events is that state objects may be created to conditionally delay the values of inputs to an equation.

There are a number of other issues involved in time-based reasoning, simplification, and the handling of multiple representations, which cannot be presented here due to space limitations. Interested readers will find these discussed more fully in [16].

AN EXAMPLE.

Consider the simple example shown in figure 2, contrived to demonstrate the creation of state objects and the collapsing of intermediate structure. For this example we will model two drive strengths, :DRIVE and :Z. (:Z means "undriven".) We will model four signal values, 1, 0, :X, and :*. (":*" means "no-value", and is used to model a high-impedance gate.) The transmitted signal is a value vector "(vv <value> <drive>)".

The SIMMER (LISP) program code for the clocked inverter functional model is shown in figure 3. This code is converted to the event representation as described above. The events are an explicit standardized encoding of the transitions implicitly specified by the program code, as shown in figure 4. Each variable name represents a bus port or a state object. Events are described from the point of view of the circuit object generating them, and represent its response at any invocation.

Next the event representation is converted to an algebraic temporal equation-based form, as shown in figure 5. Variables now represent (port or state object) timelines indexed by time-points. Automatic simplification performed prior to substitution has removed the extraneous clauses (such as "`(AND T ...)`" in figure 4) introduced by the symbolic execution.

The equations from all the blocks and busses in the circuit are successively composed, alternating with simplification to reduce the combinatorics of intermediate forms. The resulting code produced for the example circuit is in figure 6.

As an experiment in functional abstraction, functional models were written for the primitive devices of clocked and unclocked inverters, Nand and Nor gates, non-inverting tri-statable drivers and latches, and a one-bit RAM cell. These were used to compose SCORE cell library functional models, and also assembled into a full 2901 bit-slice plus control logic. Table 1 details some results of abstracting netlist-level circuits (netl) to a single functional model

```
(DEFUN PAPER-EXAMPLE-FCN (IN OUT ENA)
  "generated by computer from circuit structure"
  (DEPENDS-ON 'NIL 'NIL
    (DRIVE-BUS IN SELF '(BIT) (VV :* :Z) 0.0))
  (DEPENDS-ON 'NIL 'NIL
    (DRIVE-BUS ENA SELF '(BIT) (VV :* :Z) 0.0))
  (DEPENDS-ON '(ENA) 'NIL
    (COND ((NOT (LOGZERO (READ-BUS ENA '(BIT))))
           (PUT-MY-STATE SELF '(ENA-483)
             (READ-BUS ENA '(BIT)) 0.0))))
  (DEPENDS-ON '(ENA IN) 'NIL
    (COND ((NOT (LOGZERO (READ-BUS ENA '(BIT))))
           (PUT-MY-STATE SELF '(IN-485)
             (READ-BUS IN '(BIT)) 0.0))))
  (DEPENDS-ON 'NIL '((ENA-482) (IN-484))
    (DRIVE-BUS OUT SELF '(BIT)
      (VV (COND ((LOGONE
                   (GET-MY-STATE SELF '(ENA-482)))
                 (LOG-NOT
                   (GET-MY-STATE SELF '(IN-484))))
                (T :X))
          :DRIVE)
      0.0))
  (DEPENDS-ON 'NIL '((IN-485))
    (PUT-MY-STATE SELF '(IN-484)
      (GET-MY-STATE SELF '(IN-485)) 2.7))
  (DEPENDS-ON 'NIL '((ENA-483))
    (PUT-MY-STATE SELF '(ENA-482)
      (GET-MY-STATE SELF '(ENA-483)) 2.7)))
```

Figure 6. The program code generated for the example circuit of figure 2. The clocked inverter delay is 1.7 and the unclocked inverter delay is 0.5, yielding 1.0 for the final unclocked inverter pair. DEPENDS-ON suppresses activity unless one of the bus list or state list values has changed.

(fcnl). Simulation data is for identical inputs during the same simulated time period, running SIMMER on a Symbolics LISP machine. Because the project goals were mainly to demonstrate the concept, not to optimize the performance of the code module produced, the data should be taken as indicating trends only. Even so, the performance improvement is clear and substantial.

Since the functional models were generated only from the structure of component primitives they are guaranteed to be true to *the device as designed;* and since the process can be completed in a day, it is inexpensive and available quickly.

Circuit	No. of Blocks	No. of Busses	FUNSTRUX Time
Example of figure 1 (netl)	3	2	—
" (fcnl)	1	0	10 sec.
D-Flip-Flop w/S, R (netl)	7	5	—
" (fcnl)	1	0	85 sec.
2901 μP 1-bit slice (netl)	370	365	—
" (fcnl)	1	0	7 hrs.

Circuit	SIMMER Time	Events Processed	Function Invocations
Example of figure 1 (netl)	47 sec.	1412	606
" (fcnl)	11 sec.	873	413
D-Flip-Flop w/S, R (netl)	50 sec.	5883	1404
" (fcnl)	34 sec.	2432	846
2901 μP 1-bit slice (netl)	1083 sec.	16303	6376
" (fcnl)	287 sec.	2055	690

Table 1. Some timing results for netlist (netl) and abstracted (fcnl) circuits.

DISCUSSION: MULTIPLE PARADIGMS OF FUNCTION AND BEHAVIOR.

Every attempt to model the real world is necessarily an approximation, and every approximation rests on a number of underlying assumptions. Let us call the set of assumptions underlying a related group of device functional models, a "functional paradigm". Often the functional paradigm is imposed by the execution semantics of the simulator on which the device models are run.

One of the first realizations of our research was the great multiplicity of distinct, but coherent and conceptually adequate, functional paradigms. Interactions must be carefully considered to insure that the abstract functional model produced is functionally equivalent to the behavior of the circuit components in the simulator. For example, many different bus models have been proposed: global variable, wired OR, RC tree, uni-/bi-directional transmission line, delay line, and many others. The choice of bus model must be consistent with the underlying simulator semantics, and the device models must be consistent with both of these. The process of functional abstraction must be consistent with all three simultaneously. Examples of other points to consider include:

- The notion of "function" can be understood in several ways.
- Time might be modeled as continuous (infinitely divisible) or as discrete (some minimum time-step size).
- The handling of values at the endpoints of intervals on timelines might adopt a convention of $(a,b]$, or $[a,b)$, or simply encode the transition in the instantaneous value. (In FUNSTRUX, the event representation uses a $[a,b)$ formalism and the equation representation uses a $(a,b]$ formalism, to facilitate manipulation.)
- There are several different ways that future (pending) event en-/dequeual might be handled.
- Zero-delay elements can be modeled in any of several ways; they have no physical reality, but are often convenient.
- There are several rational ways that appropriate state objects could be inferred.
- Ports and signal flow can be unidirectional or bidirectional.

- Signals might model the signal value only, or value plus drive strength, or some other combination.

This list is indicative rather than exhaustive. The point is that there are several closely-related but alternative functional paradigms, each internally coherent, and each consistent with different underlying simulator behavior.

SHORT-COMINGS, LIMITATIONS, AND FUTURE WORK.

Zero-delay loops in the circuit become ill-constrained, because a component's behavior can then conceptually contribute to determining its own behavior. Currently we disallow all zero-delay loops (as do most simulators). Also, in the case of a pass transistor network (passive steering logic) we currently do not correctly model global network properties such as drive strength. This situation does not arise in the standard-cell circuits we considered. The restrictions we imposed on the class of circuits considered are:

- Busses connect only to blocks.
- Busses change state only when directly driven by a block.
- Zero-delay loops are disallowed (this excludes zero-delay bidirectional elements as a special case).

Extensions beyond this restricted case are a very challenging open problem for future research.

The creation of state objects is still rather unsatisfactory. We have explored dependency-graph analysis methods, simulated annealing, Markov walks, and more ad hoc approaches. All tend to result in the creation of more state objects than strictly necessary. Optimal state object creation is more general than the problem of deleting the minimum number of graph nodes to break all feedback cycles, which is known to be computationally intractable (it reduces to Feedback Vertex Set [12]). So are the subsidiary problems of identifying the node through which the most simple cycles pass, or even counting how many distinct simple cycles pass through a given node (both reduce to

Kth Shortest Path [13]). Perhaps a careful analysis of how human designers allocate state objects, or an approach like "plunking" [7], would lead to a better algorithm.

The code produced has not been optimized for human readability, but rather is organized in a way that reflects the functional composition. This is undesirable because we would like to be able to easily inspect the higher-level code produced in order to help debug faulty circuit designs.

Because a fundamental aspect of this work is that it abstracts functionality directly from the structure of the circuit, it can not be used to produce a functional model of a circuit until after the circuit structure has been designed. However, if a top-down circuit design methodology is employed intermediate functional models could be abstracted from intermediate-level components.

We have not yet incorporated error detection and exception reporting mechanisms. This could easily be done by including dummy components ("monitors") in the circuit, connected to the signals to be checked, with a functional model which called some function to check the signals and report as appropriate. Because FUNSTRUX simply passes unknown functional forms directly through as a "black box", any such special-purpose error functions would simply be composed with the overall abstract functional model, whenever they were encountered in the circuit.

This work is, of course, preliminary. Future work will involve trying to extend the abstraction mechanisms reported here to encompass a wider range of functional simulation languages. We intend to attempt at least one commercial (production-quality) functional simulator, and to automatically generate simulator models for it from primitive elements. We would like to recognize higher-level roles from low-level implementations. For example, FUNSTRUX will correctly compose the functional models of the components of an adder to produce a higher-level model which correctly models the adder, bitwise, but cannot recognize that a simple integer "+" could be directly substituted.

Another desirable area would be the ability to incorporate simplifying assumptions about the surrounding circuitry or signals.

RELATED WORK.

The critical need to verify the functional correctness of complex designs is well known. Simulation is an important part of practical correctness verification, so much so that there are over 100 languages for simulation and behavior [9]. Several companies produce functional simulation models commercially [4, 17]. Various temporal-based systems attempt to capture the time behavior of circuits using logical axioms or equations [1, 18, 23, 25, 26]. Schwartz and colleagues [25, 26] used the $\Leftarrow$ temporal operator to reason about communication protocols. Hunt and Moore [18] used temporal logic to verify a microprocessor. Amblard et al. [1] proposed an equivalent representation to our equations for reasoning about circuits, but did not implement the method and did not relate it to simulation. Meinen [23] used a notation similar to *our enablement predicate* (:EP), showed that every group of edge-controlled transfers can be written as a group of interval-controlled transfers and vice versa, and proposed that an event-based simulation could be automatically constructed from an extracted circuit. To our knowledge, however, our work is the first to produce executable functional model program code for a circuit automatically from the executable program code of the circuit components.

This research also contributes to systems which try to understand how circuits work. It is a start at understanding the analytical knowledge necessary to complement Hall's work on learning [14, 15]. Kelly and Steinberg [19] explored functional composition using timelines. Mitchell [24] showed how a functional explanation could be used to learn design rules in a design grammar. Sussman [28] advanced the notion of multiple representations of sub-circuits and components.

Several hardware diagnostic systems have combined structure and function [6, 8]. These systems are usually directed at the board level rather than

VLSI circuits. Shirley [27] used structure and behavior information to design tests for VLSI. Darlington [5] used a transformational approach to optimizing program code which is conceptually similar to our syntactic simplifications.

SUMMARY AND CONCLUSIONS.

The behavior of a circuit is implicit in the behavior of its parts and their structural connections. We have investigated the abstraction of function from structure in VLSI, and mapped out a mechanism for doing this with one functional simulator. A LISP prototype (FUNSTRUX) was implemented to experimentally investigate this possibility. Input is the program code for the functional models of the circuit components, together with a netlist-like description of their connectivity. Output is the program code for a single functional model of the circuit as a whole. FUNSTRUX was used to automatically create SIMMER functional models for the SCORE standard cell library. We also used FUNSTRUX to generate a functional model for a 2901 microprocessor 1-bit slice plus control.

Some of the key points in this functional model abstraction process include:

- Knowledge about behavior and time is encoded in specialized representations suitable for specialized manipulations: in LISP code, in algebraic temporal equations, and in an event formalism.
- Program code semantics are encoded by the translation to and from the event formalism.
- The event formalism makes explicit and manipulable the signal transitions implicit in the program code.
- Temporal operators and equations facilitate substitution and composition, as well as reasoning about time dependencies.
- A rule-based system is used for simplification of expressions, which can be easily customized for specialized syntax.

- Unknown forms and procedures inserted in the code are treated according to the semantics of pure LISP, easily accommodating special-purpose functions written by a designer.
- State objects and accessors are inferred as necessary.
- Dependency analysis allows conditionalized suppression of simulator activity unless values may have changed.

We hope that this approach captures important features of many event-driven simulation models, and so will be reasonably general as we go to other simulators/languages. Ideally, a wide variety of event-driven simulators will be expressible in an event notation in a way that is largely independent of particular coding conventions of particular programmatic function-describing languages and simulators. This would permit much of the system machinery to be re-targeted at a new simulator by simply re-writing the code-based to event-based translator, rather than redeveloping a whole new system.

ACKNOWLEDGMENTS.

The authors would like to acknowledge helpful discussions with Mark Alexander, Walter Hamscher, Chuck Rich, Brian Williams, and Patrick Winston, assistance from Ron Rivest on the problem of creating state objects, and helpful comments from the anonymous reviewers. Personal support for the first author was furnished by an IBM Graduate Fellowship, and during the early stages of this research by an NSF Graduate Fellowship. Personal support for the second author was furnished by an NSF Graduate Fellowship. This paper was prepared jointly at the Gould Semiconductor 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.

REFERENCES.

[1] Amblard, P., Caspi, P., and Halbwachs, N., "Describing and Reasoning about Circuits Behavior by Means of Time Functions", in Proc. of 7th Int'l IFIP Symposium on Computer Hardware Description Languages and Their Applications, 1985.

[2] Bipolar Microprocessor Logic and Interface, Am2900 Family 1985 Data Book, Advanced Micro Devices, 1985.

[3] Alexander, Mark, "A Spatial Reasoning Approach to Cell Layout Generation", IEEE 1986 Custom Integrated Circuits Conference (CICC'86), May, 1986, pp. 356-359.

[4] Bloom, M., "Behavioral Models Take the Pain Out of System Simulation", *Computer Design,* 15 Feb., 1987, pp. 38-46.

[5] Darlington, J.; "An Experimental Program Transformation and Synthesis System"; *Artificial Intelligence* 16 (1981), 1-46; North-Holland Publishing Company.

[6] Davis, Randall; Shrobe, Howard; "Representing Structure and Behavior of Digital Hardware", *Computer,* vol. 16, no. 10, Oct. 1983, pp. 75-82.

[7] de Kleer, Johan; and Brown, J.S.; "Theories of Causal Ordering;" *Artificial Intelligence* 29 (1986) pp 33-61.

[8] de Kleer, Johan; Williams, Brian; "Reasoning About Multiple Faults", Proc. Fifth Natl. Conf. on Artificial Intelligence (AAAI'86), Philadelphia, Pa., 11-15 August 1986, pp. 132-139.

[9] Dewey, Al; "The VHSIC Hardware Description Language (VHDL) Program", 21st IEEE Design Automation Conference (DAC'84), Albuquerque, NM, 25-27 June 1984, pp. 556-557.

[10] Drivian, Roy L.; "Functional Models for VLSI Design", 20th IEEE Design Automation Conference (DAC'83), 1983, paper 32.2, pp. 506-514.

[11] Gajski, Daniel D., Kuhn, Robert H., "Guest Editor's Introduction: New VLSI Tools", *Computer,* December 1983, pp. 11-14.

[12] Garey, M. R., and Johnson, D. S., *Computers and Intractability; A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1979, p. 191.

[13] ibid., p. 214.

[14] Hall, Robert J.; "Learning by Failing to Explain", M.I.T. Artificial Intelligence Laboratory Technical Report 906, Cambridge, Ma., May 1986.

[15] Hall, Robert J.; "Learning by Failing to Explain", Proc. Fifth Natl. Conf. on Artificial Intelligence (AAAI'86), Philadelphia, Pa., 11-15 August 1986, pp. 568-572.

[16] Hall, Robert J.; Lathrop, Richard H.; and Kirk, Robert S.; "A Multiple Representation Approach to Understanding the Time Behavior of Digital Circuits", to appear, Proc. Sixth Natl. Conf. on Artificial Intelligence (AAAI'87), Seattle, Wa., 13-17 July, 1987.

[17] Hiserote, Jay, Morris, James B., and Hunter, Robert D., "Semicustom IC Simulations on CAE Workstations", *VLSI Systems Design,* December 1985.

[18] "Mathematical Proof Verifies Error-Free Processor Design", *Electronics,* May 26, 1986, pp. 36-37.

[19] Kelly, Van E.; Steinberg, Louis L.; "The CRITTER System: Analyzing Digital Circuits by Propagating Behaviors and Specifications", Proc. Natl. Conf. on Artificial Intelligence (AAAI'82), August 18-20, 1982, C.M.U., U. of Pittsburgh, Pittsburgh, Penn., pp. 284-289.

[20] Kirk, Robert S; Hall, Robert J.; and Lathrop, Richard H.; "SCORE Cell Development Environment", IEEE 1987 Custom Integrated Circuits Conference (CICC'87), May, 1987.

[21] Lathrop, Richard H.; Kirk, Robert S.; "An Extensible Object-Oriented Mixed-Mode Functional Simulation System", 22nd IEEE Design Automation

Conference (DAC'85), Las Vegas, Nev., 23-26 June 1985, paper 39.2, pp. 630-636.

[22] Lathrop, Richard H.; Kirk, Robert S.; "Precedent-based Manipulation of VLSI Structures", 23rd IEEE Design Automation Conference (DAC'86), Las Vegas, Nev., 29 June - 2 July 1986, paper 38.5, pp. 667-670.

[23] Meinen, P., "Formal Semantic Description of Register Transfer Language Elements and Mechanized Simulator Construction", Proc. of 4th IEEE Int'l Symposium on Computer Hardware Description Languages (1979).

[24] Mitchell, Tom M.; Mahadevan, Sridhar; Steinberg, Louis I.; "LEAP: A Learning Apprentice for VLSI Design", Proc. 9th Intl. Joint Conf. on Artificial Intelligence (IJCAI'85), Los Angeles, Ca., 18-23 Aug. 1985, vol. 1, pp. 573-580.

[25] Schwartz, Richard L.; Melliar-Smith, P. Michael; "From State Machines to Temporal Logic: Specification Methods for Protocol Standards", *IEEE Trans. on Communications,* vol. COM-30, no. 12, December 1982, pp. 2486-2496.

[26] Schwartz, R. L.; Meliar-Smith, P. M.; Vogt, F. H.; Plaisted, D. A.; *An Interval Logic for Higher-Level Temporal Reasoning,* NASA Contractor Report 172262, Contract NAS1-17067, September 1983.

[27] Shirley, Mark H., "Generating Tests by Exploiting Designed Behavior", Proc. Fifth Natl. Conf. on Artificial Intelligence (AAAI'86), Philadelphia, Pa., 11-15 August 1986, pp. 884-890.

[28] Sussman, Gerald J.; "Slices: at the Boundary Between Analysis and Synthesis", in *Artificial Intelligence and Pattern Recognition in Computer Aided Design,* J.-C. Latombe (ed.), North-Holland Pub. Co. (Amsterdam), 1978, pp. 261-299.

END
9-87
DTIC