Advances in Functional Abstraction From Structure

Richard H. Lathrop*, Robert J. Hall*, Gavan Duffy**, K. Mark Alexander***, Robert S. Kirk***

* MIT Artificial Intelligence Laboratory, Cambridge, MA
** University of Texas, Austin, TX
*** Gould Semiconductor Division, Pocatello, ID

ABSTRACT: FUNSTRUX has been extended to extract behavioral level models for a commercial simulator directly from a circuit netlist. Recent advances include: a retargetable code generation mechanism; an object-oriented control structure; handling of initialization values; and improved run-time and space requirements of the abstraction process. We discuss some of the issues that arise in translating from LISP to 'C' and from one functional paradigm to another.

I. INTRODUCTION.

FUNSTRUX is a functional abstraction system that automatically produces a functional simulation model from a netlist input. FUNSTRUX effectively transforms the netlist into computer program code suitable for execution by a functional simulator. Previous discussion [5, 6, 8] introduced a theory and prototype implementation of functional abstraction from structure. This paper describes recent advances that explore independence of target implementation language. These advances allow the FUNSTRUX prototype to produce 'C' code for the Mentor QUICKSIM (TM) simulator [11]. We discuss issues arising in translating from LISP to 'C' and one functional paradigm to another.

FUNSTRUX generates a circuit functional simulation model by transforming the simulation program code for the circuit's low-level components into equations in a temporal (time-based) algebra, combining the equations according to the circuit netlist, and transforming the result back into program code. The process produces a functional simulation model that remains faithful to the circuit as actually designed.

Advances described in this paper now enable FUNSTRUX to output functional simulation models written in 'C', targeted for the Mentor QUICKSIM (TM) commercial simulation system. Input functional simulation models for the circuit components are still written in a subset of SIMMER (LISP) code [7, 8]. Thus the process involves translation of both the source code and the functional paradigm.

Section II outlines the process of functional abstraction. Section III describes FUNSTRUX architecture modifications needed to re-target the output from LISP to 'C'. We discuss differences in both source code language and the underlying functional paradigms of QUICKSIM and SIMMER. Section IV treats the explicit representation of initial values, from the temporal algebra to the generated code. Section V presents an efficient DAG representation of the temporal equations generated. Section VI reviews current limitations, shortcomings, and future work.

II. FUNCTIONAL ABSTRACTION FROM STRUCTURE.

FUNSTRUX is not a simulation language. It is a system that manipulates functional simulation languages according to circuit structure. Technical details are contained in references [6, 8].

The complex functionality of VLSI circuits arises from the interconnection of many simple components. Functional abstraction captures the functionality of VLSI circuits while abstracting away (merging) many low-level details. The simple behavior of each low-level component is represented in a context-independent manner. These low-level component functions are analytically combined according to the circuit netlist, producing a representation of global circuit behavior.

Inputs to FUNSTRUX are the netlist plus the program code of each component's functional simulation model, written for the SIMMER functional simulator (see figure 1). SIMMER code is essentially LISP code, with some built-in functions that read and drive busses or get and put internal state objects. Restrictions imposed on the class of circuits are that (a) busses connect only to blocks, (b) busses change state only when directly driven by a block, and (c) zero-delay loops are disallowed.

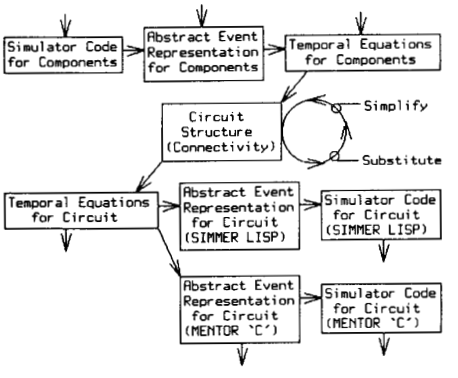

Figure 1. The FUNSTRUX system architecture. Program code for the functional models of the circuit components is converted to equations in a temporal algebra, substituted and simplified according to the circuit structure, and converted back to functional model program code for the whole circuit.

FUNSTRUX first symbolically executes each component's functional simulation model code to determine how output events would depend on the values of inputs and internal state objects. The symbolic output events thereby produced are next converted to temporal algebra formulae. These describe the time-varying values of outputs and internal state as algebraic functions of the time-varying values of inputs and internal state. Algebraic formulae can be easily combined by substitution of equals for equals. In a circuit, two values are equal if they occur at the same point.

FUNSTRUX next walks through the circuit structure given by the netlist. At each step it substitutes one function-describing algebraic equation into another. To avoid combinatorial explosion in expression size, pattern-action rules simplify expressions at each step. A global set of algebra formulae result. These describe the time-varying values of circuit-level outputs and state objects as algebraic functions of the time-varying values of circuit inputs and state objects. The formulae are finally converted back into program code.

Figure 1 depicts the FUNSTRUX system architecture. The arrows at top and bottom emphasize that behavioral descriptions could in principle enter or leave at any of the behavioral representations. In practice, we have found it useful to inspect the global circuit equations in order to help debug circuit functionality.

The FUNSTRUX code abstraction process differs from the levelized compiled code approach [15]. Levelized compiled code works only for synchronous machine design. FUNSTRUX works for general digital circuits and handles both synchronous and asynchronous logic. Being targeted at the functional model level, FUNSTRUX differs from compiled logic-level simulation [15, 3].

III. RE-TARGETING THE ARCHITECTURE.

Initially, FUNSTRUX generated LISP code for the SIMMER simulator. In order to generate models suitable for the QUICKSIM simulator it is necessary to generate 'C' code. We chose to continue using LISP as the internal representation and to convert it to 'C' as a final post processing step (example of 'C' code produced omitted due to space limitations).

There are several areas of concern to be considered with respect to the code generation process:

- Organization of the FUNSTRUX control structure to permit easy re-targeting of the program code output;
- Syntactic and semantic differences between LISP and the target simulator's native language;
- Semantic differences between FUNSTRUX event scheduling and the target simulator's event queue handling; and
- Differences in the underlying functional paradigm between FUNSTRUX and the target simulator.

FUNSTRUX Control Structure.

FUNSTRUX manages its translations to and from equations, events, and code using object-oriented programming techniques. Typed data structures organize all aspects of the process. Atop the hierarchy sits a FUNSTRUX program object. Directly beneath it are "model" managers particular to any modeled device. Under the model managers are one or more behavior managers. These direct various behavior representations (currently code, events, and equations) of the modeled device. Event and equation managers can be shared across behavior managers within any one model manager.

This supports the construction of functional simulation models for different simulators using the same event models and equations. These objects inherit abstract objects that maintain data (e.g., event managers and equation managers) and perform operations (e.g., event-to-equation and equation-to-event translation) that are common across simulators.

LISP and the Target Language.

Syntactic differences between LISP and the target language are handled in LISP as any other symbolic computation. LISP operates on program language constructs as easily as on any other symbolic data. Thus, for example, the fact that LISP is a prefix-operator language while 'C' is an infix-operator language is largely irrelevant. ELLA [9, 10] provides an interesting example of an existing language allowing expressions to be stated either imperatively (as in 'C') or applicatively (as in LISP).

Semantic language differences were for the most part handled by coding a compatibility package in 'C'. The only major difference involved operations such as IF, which return a value in LISP and none in 'C'. This was handled by a syntactic translation to emulate the corresponding side-effect in 'C':

```
;;; LISP form:
          (setq foo (if bar 10 20))
;;; Syntactic Translation:
          (if bar (setq foo 10) (setq foo 20))
;;; 'C' form:
          if (bar) {foo = 10} else {foo = 20};
```

The 'C' compatibility package implements language abstractions that allow the expression of common behavioral motifs of digital hardware. Care was taken in writing these routines so that they could be ported to other simulators.

FUNSTRUX Event Scheduling.

Although simulation is "event driven", not all simulators handle event processing in the same way. For example, ELLA's unusual timing paradigm reverses the usual ordering and refers to past, rather than future, events. There is no guarantee that the event scheduling paradigm assumed in FUNSTRUX would translate easily to any given simulator's queue handling. An example was encountered when targeting the QUICKSIM functional simulator. QUICKSIM does not allow multiple simultaneously pending events for the same output port at different future times. More than one pending event for an output port is considered a spike, a warning is generated, and the other events are ignored. SIMMER maintains, and FUNSTRUX exploits, a full pending event queue that schedules these independently.

Engineering trade-offs center around efficiency and portability. An implementation highly customized to the particular features of a target system could exploit these to gain execution speed, but substantially more engineering effort would be needed to re-target to each new system. Because (a) well over a hundred different hardware behavior description languages currently exist [4], (b) better ones are constantly being devised, and (c) FUNSTRUX is still a prototype system, we opted for flexibility at the expense of efficiency. We find that most of the time savings generated by collapsing redundant circuit structure has been consumed by the overhead of a queue mechanism foreign to the target simulator (data omitted due to space limitations). At this stage we are not yet attempting to produce optimal 'C' code, and no critical routines are written in assembly language.

A subroutine library written in the target simulator language allows FUNSTRUX to control the functional simulation model's future internal state events. FUNSTRUX generates target simulator code that creates and manipulates an event-handling data structure, designed to maximize portability across simulators without significantly depending on the queue mechanism of the target simulator. Requirements on the target architecture are limited to (a) the ability to store complex data structures as state objects, and (b) the ability to schedule a functional simulation model re-invocation at some point in the future.

Differences in the Functional Paradigm.

Every attempt to model the real world is necessarily an approximation, resting on a number of underlying assumptions called a "functional paradigm", as discussed in [8]. Often the functional paradigm is imposed by the execution semantics of the simulator on which the device models are run. Different functional paradigms may have different models of time (continuous or time-step), different modeling of zero-delay elements, different bus models (global variable, wired OR, delay line), and so forth. In some cases, these may cause different simulators to produce different output values for the same circuit under the same input stimuli.

One example of differences in functional paradigm between the QUICKSIM and the SIMMER simulators is seen in circuits with cascaded zero-delay elements in which different zero-delay paths converge and their zero-delay path-lengths differ. For example, when such a situation drives a gated direct-action device (perhaps the clock of an edge-triggered latch), QUICKSIM's functional paradigm sometimes allows the latch to trigger, even though the latch's clock bus records flat and no glitch warning is generated. In contrast, SIMMER implements a roll-back mechanism that insures that the final value of all cells reflects the final value of all zero-delay paths, regardless of length (i.e., as if all zero-delay changes had occurred instantaneously and correctly).

FUNSTRUX' current time formalism does not allow the representation of multiple signal values at the same point at the same time. This makes it difficult to model QUICKSIM's zero-delay propagation in all cases, because FUNSTRUX cannot represent the first such value as distinct from the last. Thus in some cases having to do with zero-delay anomalies, the behavior of the FUNSTRUX-generated model may diverge.

IV. EXPLICIT REPRESENTATION OF INITIAL VALUES.

Circuit designers must design a circuit so that it can be forced into a correct known state at initialization. For example, before a flip-flop is clocked, its output is unknown. This is indicated in the code and event representations by having an initial event (at start of simulation) which sets the internal node to :X (or, any desired value). This is reflected in temporal equations by a conditional, which states that "if the clock has ever been true, then output is the normal flip-flop behavior; otherwise, output is :X."

```
output(t) =
  (if (> ({⇐ (u)  (clock-true(u))  (t)})
         minus-infinity)                    ;test
      input({⇐ (u)  (clock-true(u))  (t)})  ;normal
      :X)                                   ;uninitialized
```

The expression ({⇐ (u) (clock-true(u)) (t)}) denotes [6, 12, 13] the most recent time since t that the predicate `clock-true` was true, or minus infinity if never true. Thus the left-arrow expression is greater than minus infinity if and only if the clock has ever been true, in which case the output has a normal (initialized) value.

A similar problem arises for the output of a gate with fixed delay. What is its output for the first gate delay after the simulation starts? We can, if we desire, again handle this by code that sets the nodes to :X at simulation start. In temporal equations we test whether t is greater than the gate delay. In the code this is translated into a run-time check.

V. AN EFFICIENT REPRESENTATION OF THE EQUATIONS.

The time and memory usage of the substitution and simplification algorithms constituted a major performance bottleneck in the original implementation of FUNSTRUX. Two factors conspired. First, functional expressions were represented using binary trees (LISP symbolic expressions), thus causing identical subexpressions to be represented once per instance. Second, the functional expressions resulting from circuits were highly redundant, due in part to fanout of gates and in part to the temporal algebra formalism.

We therefore changed the internal representation of functional expressions to a maximally shared DAG (Directed Acyclic Graph). DAGs are a well-known [14] tool for representing VLSI structure. However, we applied this notion to the representation of circuit function.

DAGs are efficient for substitution and simplification because any identical subexpressions are represented by pointers to a single copy of that subexpression. The procedures we developed perform the destructive mutations required for substitution and simplification in a number of operations roughly proportional to the depth of the expression, not to the size of the whole expression. This representation change improves time and space performance dramatically.

VI. LIMITATIONS, SHORT-COMINGS, AND FUTURE WORK.

A number of further abstractions in our modeling of time would be useful. A general strategy for handling classes of operators that transform the time-line is desirable. Creating stable-changing-stable intervals (possibly in conjunction with inertial delays) would help collapse multiple minor delays in the time-line. This would let outputs of combinational logic, counters, etc. become stable at the same time, not over a small interval. Some aspects of the levelized compiled code [15] might also be useful to include.

We would also like to improve our treatment of time-points in the temporal algebra, by permitting their joint comparison and arithmetic combination (currently time-points can only be compared or combined with constants). This would permit a cleaner modeling of such cases as conditionally varying rising and falling signal delays, which are possible but awkward in our current version of time handling. Non-atomic time interval endpoints could represent a wider class of behaviors.

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). Convergent zero-delay paths of different lengths cause problems in modeling QUICKSIM's glitch handling.

We have not yet explored optimization of the generated simulator code, and further optimizations could be performed directly on the generated source code. The code is also not organized for human comprehensibility, but reflects the circuit structure. This is undesirable because we would like to be able to inspect easily the higher-level code produced in order to help debug faulty circuit designs. The designer should be able to specify individual circuit nodes as state objects, so that their values could be traced. Currently FUNSTRUX' state object creation mechanism is completely automatic.

Although the syntactic translation from LISP to 'C' was straight-forward, we have discovered several differences in the underlying functional paradigms of SIMMER, FUNSTRUX, and QUICKSIM. Some differences were resolved by introducing special constructs, such as the internal queue. The simulated behavior of convergent zero-delay paths with different path-lengths is a functional paradigm difference that cannot be resolved in this way.

The attempt to build a general functional abstraction engine involves both syntactic and semantic language and paradigm issues in the match to the target simulator. We have found the syntactic language issues easiest to resolve. Differences in the functional paradigm have proven more stubborn. Their resolution sometimes involves implementing additional machinery (and hence efficiency loss), or functional models whose glitch behavior does not match QUICKSIM's.

VII. SUMMARY.

We have described recent advances in functional abstraction from structure. These included a direct representation of starting conditions in the temporal algebra, which yields a cleaner handling of initialization values, and an improved internal representation of the temporal equations resulting from circuit structure. We have described the LISP to 'C' conversion process, and discussed issues involved in differences between functional paradigms. We have used FUNSTRUX to produce 'C' code for the Mentor QUICKSIM (TM) simulator, starting from a circuit netlist and program code for the circuit components written in a subset of LISP for the SIMMER simulator.

ACKNOWLEDGMENTS.

The authors would like to acknowledge helpful discussions with Walter Hamscher, Chuck Rich, Brian Williams, and Patrick Winston, and assistance from Ron Rivest on the problem of creating state objects. 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] "Bipolar Microprocessor Logic and Interface, Am2900 Family 1985 Data Book", Advanced Micro Devices, 1985.

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

[3] Bryant, R. E., Beatty, D., Brace, K., Cho, K., Sheffler, T., "COSMOS: A Compiled Simulator for MOS Circuits", 24th ACM /IEEE Design Automation Conference (DAC'87), Miami Beach, FLA, 1987, pp. 9-16.

[4] 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.

[5] Hall, Robert J.; "A Fully Abstract Denotational Semantics for Event-based Simulation", Proc. IASTED *Intl. Symposium* Applied Simulation and Modeling (ASM'87), Santa Barbara, CA, 27-29 May, 1987.

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

[7] Lathrop, Richard H., and Kirk, Robert S., "An Extensible Object-Oriented Mixed-Mode Functional Simulation System", 22-nd ACM/IEEE Design Automation Conference (DAC'85), Las Vegas, Nev., 23-26 June 1985, paper 39.2, pp. 630-636.

[8] Lathrop, Richard H.; Hall, Robert J.; and Kirk, Robert S., "Functional Abstraction From Structure in VLSI Simulation Models", 24th ACM/IEEE Design Automation Conference (DAC'87), Miami Beach, FLA, 1987, pp. 822-828.

[11] "QUICKSIM Family Reference Manual", Mentor Graphics Corporation, 1987

[9] Morison J. D., Peeling N. E., Thorp, "A Database Apporoach to Design Data Management and Programming Support for ELLA, a High-level HDDL", IFIP 7th International Symposium on Computer Hardware Description Languages and their Application, CHDL '85, Tokyo, Japan, 1985.

[10] Morison J. D., Peeling N. E., Thorp, "The Design Rationale of ELLA, a Hardware Design and Description Language", IFIP 7th International Symposium on Computer Hardware Description Languages and their Application, CHDL '85, Tokyo, Japan, 1985.

[12] 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.

[13] 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.

[14] Sedgewick, R., *Algorithms,* pp 426–428, Addison-Wesley, Reading, MA, 1983.

[15] Wang, L.-T., Hover, N. E., Porter, E. H., and Zasio, J. J., "SSIM: A Software Levelized Compiled-Code Simulator", 24th ACM/IEEE Design Automation Conference (DAC'87), Miami Beach, FLA, 1987, pp. 2-8.