METAPROGRAMMING IN DIGITAL SIMULATION

J.W. Eicher
Air Force Phillips Laboratory
Hanscom AFB, Massachusetts

F.M. Brown
Air Force Institute of Technology
Wright-Patterson AFB, Ohio

Abstract

Many computing tasks could be carried out more efficiently if programs could tailor themselves to the data they process. Such adaptation is difficult to program in procedural languages. The non-procedural languages LISP and Prolog, however, treat programs as data; as a result, metaprograms may be employed in these languages to construct, analyze, and modify application-programs. This paper describes the use of metaprogramming in Prolog to construct digital logic-simulators which adapt themselves for efficiency to the structure of the circuits they simulate.

To test the utility of the metaprogramming approach to simulation, in a simple but representative domain, a logic-simulator was written for TTL integrated circuits. The simulator accepts a wiring-list, from which it creates, loads, and executes a Prolog rule which is both (a) a representation of the user's circuit at the level of gates, adders, counters, etc., and (b) a procedure for its simulation. This rule, which expresses the structure of the circuit by means of Clocksin's definitional format, is an example of "executable data"; there is no distinction in Prolog, that is, between executable code and data. The ordering of goals in this rule guarantees that signals are traced through the circuit with no backtracking; thus the time to execute a simulation-cycle is minimized.

1 Introduction

In standard higher-order procedural languages, such as Basic, C, Fortran, or Ada, there is total separation between code and data. Long familiarity with procedural languages makes this seem a natural and inevitable part of programming. The non-procedural languages LISP and Prolog, however, remove this separation; these languages enable one to write metaprograms [1, 7] i.e., programs that treat programs (perhaps themselves) as data.

Metaprogramming is particularly simple and direct in Prolog, which has only one data-structure, the term. The same format is used to represent both program-elements and data; data-statements are therefore "executed" in the same way as program-statements.

The blurring of program and data in Prolog also blurs the distinction usually enforced between decisions the programmer makes when writing a program and decisions the computer makes while executing the program. A metaprogram can examine itself while it is running, and modify itself in pursuit of some objective. Consider an aeronautical analogy. An ordinary "object-airplane" can accommodate in flight to varying tasks and aerodynamic conditions only by choosing from among options (flaps, rudder, ailerons, wing-angle, thrust, etc.) which a designer has provided ahead of time. A "meta-plane," on the other hand, could re-design and re-build itself in flight.

Metaprogramming for in-flight re-design of Prolog programs has been studied by Komorowski [5], Takeuchi and Furukawa [8], and others. Such re-design is particularly appropriate in the field of simulation, where it seems not to have received widespread attention.

Two kinds of data must be supplied to a digital logic-simulator: (a) a description of the structure of the circuit and (b) information concerning the current input-values. The structure of the circuit remains constant throughout the many cycles of a simulation; only the input-values change from one cycle to the next. In many cases, therefore, it is worth the computational overhead for a meta-simulator to examine a given circuit's structure, construct an object-simulator that takes optimal advantage of that structure, and execute the optimized object-simulator. This approach was proposed in [4].

This paper concerns logic-simulators, i.e., simulators in which ideal timing assumptions are made; such simulators are useful for initial design. Timing problems, arising from differential delays, must be considered, however, as design proceeds. Fortunately, as Dukes et al. [3] have shown, delay-models are readily incorporated into Prolog-based simulators.
The simulators we discuss are programmed in the Prolog language at both the meta- and object-levels. It may be more practical in some settings, however, for a Prolog meta-simulator to write an object-simulator in a procedural language such as C or Ada.

2 Clocksin's Definitional Representation

Clocksin [2] has shown how the unique character of Prolog can be exploited for the purpose of specifying and automatically reasoning about digital circuits. His "definitional representation" is a directly executable specification for the connection of components in a digital system. Like the specifications employed in many hardware description languages, the definitional representation is hierarchical and modular. Unlike the specifications written in such languages, however, specifications written in Prolog may be readily manipulated as objects.

2.1 Definitional Specification

A definitional specification of a module (either the entire circuit or one of its components) is a Prolog rule of the form shown in Figure 1. The head of the rule represents the module to be defined, and the body of the rule is a composition of its constituent sub-modules. The operator ":-" is interpreted to mean "is defined by." The component-specifications in the body of the rule may be listed in any order. Each sub-module is defined either by a rule of the same form or, if it is primitive, by a collection of Prolog facts (a fact is a rule having a head, but no body).

The definitional representation of the full-adder circuit shown in in Figure 2, for example, is the rule shown in Figure 3.

To achieve a complete specification of the full-adder circuit, the rule shown in Figure 3 must be augmented with rules defining the modules xor and nand. Let us assume that the xor-gate is composed of nand-gates, as shown in Figure 4. The associated xor-rule is given in Figure 5. We treat the nand module in this discussion as atomic; hence, it is defined by explicit Prolog facts, as shown in Figure 6, rather than in terms of more basic modules.

2.2 Advantages of the Definitional Form

Clocksin cites a number of advantages that result from specifying a circuit definitively. The module-name is explicitly part of the specification, permitting easy modular decomposition. Internal connections, which are named by variables not appearing in the head of the rule, are effectively hidden; such lexical scoping is good engineering practice. Bidirectionality, which is important in specifying the behavior of some components, such as pass-transistors, is represented inherently because Prolog deals with relations rather than functions. Although the user...
3 Design of a Meta-Level Simulator

Given a specification-format describing a circuit, the task of a meta-level definitional simulator is to examine that description, and to construct, load, and execute a suitably-tailored object-level simulator.

3.1 Steps in Meta-Simulation

An optimizing meta-level definitional simulator carries out the following sequence of steps:

1. Extract a modular description of the circuit.
2. Transform the modular description into a definitional rule.
3. Re-organize the definitional rule to remove backtracking.
4. Form the object-level simulator by converting constants denoting terminal-locations into variables denoting signal-values at those locations.
5. Execute the object-level simulator.

3.2 Example

We illustrate the foregoing steps, in a very simple setting, by showing the operation of a meta-level simulator which accepts a wiring-description of a circuit made up of 7400-series TTL IC-packages, and writes an object-level simulator at the gate/register level. In particular, we show some of its details when applied to the full-adder circuit of Figure 2.

The designer's description of the circuit is reproduced in Figure 7. Prolog facts (degenerate rules) are used to name the input and output terminals and also to indicate the connections to each of the ICs in the circuit. For example, the fact

```
package(sn7486,[1,p,2,z,3,s,4,x,5,y,6,p])
```

announces an SN7486 pin 1 of which is connected to point "p" in the circuit, pin 2 to point "z," etc. Figure 7 also includes a fact specifying a sequence of test-inputs.

The steps in meta-simulation outlined earlier are applied below to the full-adder circuit of Figure 2.

1. Extract a modular description of the circuit.
   The extraction-step begins with a low-level description of a circuit’s structure and derives a higher-level description, i.e., the level at which simulation
is to take place. In the present case, the result of extracting components from the designer’s description is a collection of facts, shown in Figure 8, asserted into the meta-simulator’s database. Each component-fact assigns a number to the component, names its type, and specifies its input and output terminals (an additional set of terminals, designating state-variables, is specified for components such as flip-flops, shift-registers, and counters—possessing internal storage).

1 A typical extraction-system might accept a CMOS mask-level description of a circuit and generate a description of the same circuit as an interconnection of transistors or, higher still, as an interconnection of components such as gates, registers, adders, and multiplexers. Because of its built-in pattern-matching and search mechanisms, Prolog is especially suited for this purpose. Dukes, et al., [3] have shown, for example, that Prolog is effective in extracting high-level behavioral and structural VHDL descriptions of CMOS-layouts comprising hundreds of thousands of transistors.

2. Transform the modular description into a definitional rule. A definitional object-level rule, shown in Figure 9, is derived from the foregoing components and asserted into the Prolog database. This rule differs from that shown in Figure 3 in several ways. First, component-numbers 1, 2, ..., 5 have been introduced for convenience into the corresponding goals. Second, an additional argument is introduced in the head to account for storage-components in the circuit; this argument is the empty list, [ ], in the present case because of the absence of such components. Finally, and most important, the arguments x, y, z, ..., c, s are in lower-case, and therefore interpreted by Prolog as constants. In Prolog terminology, the variables are “frozen.” This freezing of variables, a characteristic of metaprogramming in Prolog, is necessary because we wish in the next step to treat the variables as objects—and Prolog variables object to such treatment. The frozen arguments are manipulated in Step 3 as terminal-locations; when “melted,” in Step 4, the arguments become variables denoting signal-values.

3. Re-organize the definitional rule to remove backtracking. A modified frozen object-level rule, shown in Figure 10, is constructed from the earlier frozen rule by carrying out a topological sort [6] of the rule’s goals to ensure that each input, for each goal, is either a circuit-input or an output of a preceding goal in the rule. A sort producing this condition is guaranteed to be possible for a circuit having no combinational feedback. The resulting goal-order guarantees that signals are traced through the circuit with no backtracking; thus the time to execute a simulation-cycle is minimized.

4. Form the object-level simulator by converting constants denoting terminal-locations into variables denoting signal-values at those locations. To specify or simulate the full-adder circuit, the rule shown in Figure 10 must be “melted,” i.e., each distinct frozen argument must be converted into a distinct Prolog variable. The melted result is
circuit([x,y,z],[],[c,s]):- nand(5,[x,y],[q]), xor(2,[x,y],[p]), nand(4,[p,z],[r]), nand(3,[q,r],[c]), xor(1,[p,z],[s]).

Figure 10: Frozen object-level rule, modified to remove backtracking.

circuit([.632,.629,.626],[],[.623,.620]):- nand(5,[.632,.629],[.617]), xor(2,[.632,.629],[.614]), nand(4,[.614,.626],[.611]), nand(3,[.617,.611],[.623]), xor(1,[.614,.626],[.620]).

Figure 11: Melted object-level rule.

shown in Figure 11. The symbols .611, .614, etc., in Figure 11 denote variables expressed in Prolog's internal notation (variable-names in Prolog begin either with a capital letter or with an underscore).

5. Execute the object-level simulator. The melted object-level rule shown in Figure 11 is both a specification of the circuit's structure and also a program for simulating the circuit's behavior. Given the test-sequence in the designer's description (Figure 7), along with the identification of input and output terminals provided in that description, auxiliary code is readily constructed to feed the test-sequence to the melted object-level rule, resulting in the simulation-run shown in Figure 12.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>c</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 12: Simulation-run.

4 Summary

Metaprogramming in Prolog, based on Clocksin's definitional form, enables a logic-simulator to adapt itself to the structure of a digital circuit. This paper describes a system comprising a meta-level simulator and an object-level simulator. The meta-level simulator accepts a description of a circuit's structure (e.g., a wiring-list or mask-level description) and extracts a modular description at the desired—typically higher—level of simulation (e.g., the gate/flip-flop or transistor level). The modular description is transformed into a Prolog rule, in Clocksin's definitional form. The goals in this rule, whose arguments are "frozen," are sorted to eliminate backtracking. Finally, the arguments are "melted" into variables, forming an efficient object-level simulator.

References


