Report 83-43
Stanford -- KSL

Scientific DataLink

MARS: A Multiple Abstraction Rule-Based Simulator.
Narinder Singh,
Dec 1983

card 1 of 1

Stanford Heuristic Programming Project
Memo HPP-83-43

December 1983

MARS: A Multiple Abstraction Rule-Based Simulator

by

Narinder Singh

COMPUTER SCIENCE DEPARTMENT
Stanford University
Stanford, California 94305

MARS: A Multiple Abstraction Rule-Based Simulator

Narinder Singh

Fairchild Laboratory for Artificial Intelligence Research
4001 Miranda Ave.
Palo, Alto, CA 94304

FLAIR Technical Report No. 17
Stanford Heuristic Programming Project Memo No. HPP-83-43

December, 1983

Abstract

MARS is an experimental system that provides a framework for implementing hierarchical discrete event driven simulators. The program is independent of any domain, technology, or design methodology and has been successfully integrated with two separate design entry workstations [2], [3]. MARS takes as input a hierarchical specification of the structure and behavior of a design and performs mixed-mode simulations, different parts of a design being simulated at different abstraction levels.

The power and flexibility of the simulation environment provided by MARS has been made possible by the application of AI techniques in the explicit (symbolic) device independent representation of a design and the use of domain independent inference procedures to reason about its behavior.

CONTENTS

FIGURES

BLANK PAGE

1. Introduction

With the advances being made in the technology for fabricating integrated circuits it is now possible to build systems of unprecedented complexity. Although the fabrication of such complex systems is easily accomplished, the task of designing them is quickly becoming unmanageable. Simulation forms an integral part of the design process to verify the operations of a system. As the complexity of designs increases so does the cost of simulation and, therefore, the overall design time. To aid the design process MARS provides a flexible environment to perform efficient simulation of complex designs.

In the framework provided by MARS, simulation is viewed as a reasoning task from the structure and behavior of a device, and the initial inputs, to its state (sequence of values over time). It is appropriate, therefore, to apply the general tools developed in AI for knowledge representation and inference to the task of simulation. We have chosen MRS [5] as a foundation for providing a set of inference and knowledge representation techniques. A key feature of MRS is the ability to specify meta-level control knowledge to guide the reasoning and representation methods to the task at hand— in this case, simulation.

MARS is designed to be part of a larger system [6] whose goal is to capture and reason about the process of design (using VLSI as a test domain). Some of the tasks such a system must perform include: simulation, verification, automatic test-generation and diagnosis. In MARS the representation and reasoning procedures could have been tailored to the task of simulation, as is the case with most traditional simulators, but this would have conflicted with the goal of using a single design description to serve multiple purposes.

Using a device independent representation scheme allows the simulator to be applied to different domains, and also permits the use of arbitrary abstraction levels for the description of a design. Similarly, using domain independent inference techniques (forward-chaining, backward-chaining, resolution, etc.) simplifies the task of generating new tools to reason about a design– the creation of a new tool can be reduced to the task of specifying a set of base-level and meta-level axioms to guide existing inference methods [5].

In addition to flexibility, the application of AI techniques aids in the efficiency of simulation. The flexibility of permiting multiple-abstraction levels allows the hierarchical specification of a design. We can exploit the structural and behavioral hierarchy of a design by performing mixed-mode simulation— simulating the components of a design at the highest level possible, based on the current simulation goals.

To increase the utility of simulation, MARS provides an interactive environment for simulating a design. The results of a simulation can be displayed graphically by animating the flow of information through a design. Similar to software environments, it is also possible to trace, single-step, and define break conditions for a simulation.

The next section will describe the language for specifying the structure and behavior of a design hierarchically. Section 3 will describe the inference techniques that use these

descriptions as data in simulating a design. In section 4 we will describe how the hierarchical specification of a design is used to: automatically select the simulation level; verify a design; and simulate with partial information. Section 5 will describe the utility of hierarchy for simulation, and section 6 will describe the user interaction environment and the debugging tools provided to aid in simulating a design. Finally, the last section will provide some conclusions and areas for further research.

2. Representing a Design

For simulation, there are two important types of design information that must be recorded— structure and behavior. At the highest level, a design can be viewed as a black box with a set of ports. The ports can either be uni-directional or bi-directional, and represent the only points through which a component can interact with its environment. We can associate a behavioral description with any component, which defines the temporal relationships between the values at its ports.

We feel that it is extremely important to capture the specification of a design hierarchically to help both the designer, as an organizational aid, and the tools which will use this description as data. The hierarchical specification of a design is captured by representing the sub-components of a composite object, the interconnections of these components, and the flow of information across the hierarchy boundary. Similarly, we can hierarchically encode the behavior of a design by associating a behavioral description with all components— both primitives and composites.

In representing the structure of a design we are concerned with answering questions about the sub-components of a design and their interconnections. The structural information of a design is not time variant (over a simulation) and need not require complex processing of information to extract. It would be inappropriate to encode this structural information procedurally since procedures are best suited for the complex manipulation of information with varying environments.

In representing the behavior of a design we are concerned with encoding relationships between port values of a component. These relationships can be encoded declaratively via explicit rules whose preconditions and conclusions are propositions about the port values of a component, or they can be specified procedurally. The explicit representation of the behavior of a device permits the same description to be used for multiple purposes. By analyzing the parts of a rule we can either deduce the state of the outputs given the inputs, as in simulation, or vice-versa, as in automatic test generation. It is very difficult to analyze the parts of procedure in this manner, especially if it has been compiled. As we shall see, the description of relationships involving time is handled in a very natural way by using rules to encode behavior. In addition, a procedural specification of behavior must include extraneous information specific to the simulation task (to guide the scheduler). Also, reasoning backwards through a procedure is extremely difficult in the presence of such extraneous information.

At present we have a preliminary version of a rule-compiler which can take the general rule-based specifications and compile them into special procedures for simulation. This provides the flexibility of the rule-based approach along with the efficiency of the procedural representation.

We, therefore, feel that it is extremely important to specify the structure and behavior of a design using an explicit (symbolic) representation scheme. We have chosen to encode

these descriptions as propositions in MRS. However, it is not always convenient to specify the behavior of compiex systems via rules. MARS permits the user to specify the behavior of a component procedurally, but this should only be done when necessary.

2.1. Specifying Structure

To illustrate the specification of a design we will use the example of the device D74 shown in Figure 1. At the top level D74 has three inputs and two outputs, each of which computes a different sum of products of the inputs. If one looks one level into the sub-structure we see that D74 is made up of three multipliers and two adders. The adders themselves are made up of four full-adders, and each full-adder is composed of a collection of *and*, *or*, and *xor* gates.

The structural description of a design includes the specification of the types of the components, and the interconnections between their ports. The example below gives a partial structural description of D74 in terms of the adders and multipliers without any hierarchical information:

```
(type m1 multiplier)
(type a2 adder)
(conn  (port out m1) (port in1 a1))
(conn* (port out m2) (port in2 a1) (port in1 a2))
```

The *type* relation is used to define that a component is an instance of the specified type. For example, the statement **(type m1 multiplier)** declares that the behavior of m1 is that of a multiplier (the actual behavior must be specified separately). When a component is defined to be of a composite type its structure is instantiated completely to the primitives. For example, if we had defined the type adder to be made up of four full-adders, then each adder will have its own set of four full-adders.

Ports of components are identified by using the *port* function, which defines a particular port of the component. For example, **(port out1 d74)** identifies the first output port of d74. The *conn* relation is used to define connections between ports. The semantics of connections are not dictated by MARS— the user can define uni-directional or bi-directional connections with arbitrary delay or data transformations (see the next sub-section for examples).

The *conn** relation takes three or more arguments and allows multiple connections to be described in a single relation. For example, **(conn* a b c)** is equivalent to **(conn a b)** and **(conn a c)**.

2.2. Specifying Behavior

The behavioral specification of a device is concerned with relating values at its ports over time. In order to relate port values we must have a vocabulary for describing the value of a port at any given time. This is done using the *true* relation. The general form of this is

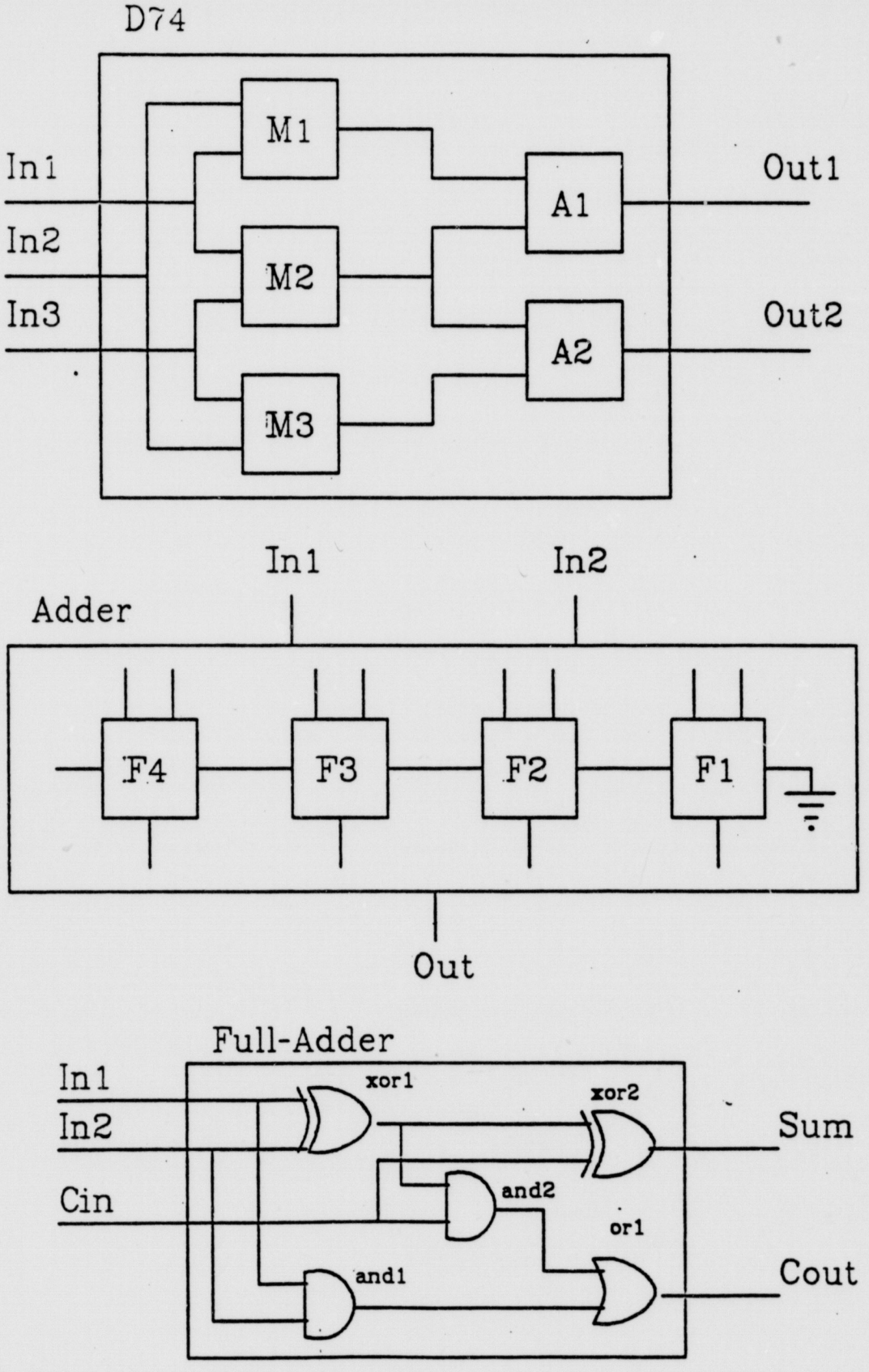

Figure 1: Hierarchical Structural Description of D74

given below:

```
(true (value (port <p> <dev>) <val>) <time>)
```

This proposition states that port **<p>** of the device **<dev>** has the value **<val>** at the time **<time>**. It is up to the user to ensure that the time units used in all the behavioral rules are mutually consistent. For example, if the time units of one component of a design are in nano-seconds, then the time units of all components must be in nano-seconds.

The specification of behavior can either be specialized for each individual component, or shared between components of the same type. The following rule defines the behavior for connections, which is shared by all connected ports. In this example all connections are uni-directional with zero delay with no data transformations. All symbols preceded by a "$" stand for universally quantified variables.

```
(if (and (conn $x $y) (true (value $x $v) $t))
    (true (value $y $v) $t))
```

The general form of a rule is **(if <precond> <concl>)**. The preconditions and conclusions are facts about the state of the device (the conclusion is not an action to be taken). When all the preconditions of a rule are satisfied the conclusions are all true. The above rule has two preconditions and one conclusion. The first precondition is true if a certain port is connected to another. The starting port of the connection is bound to the variable "$x" ,and the ending port is bound to to the variable "$y". The second precondition of the rule is true if the starting port of the connection has some value at some time. The actual value is bound to the variable "$v" and the time to "$t". The conclusion of the rule states that ending port of the connection has the same value at the same time.

The behavioral definition for connections given below reduces one level of indirection present above by specializing the connection rule. In this case a new behavior rule is specified for each connection (the conclusion of the following rule).

```
(if (conn $x $y) (if (true (value $x $v) $t) (true (value $y $v) $t))
```

State can be recorded explicitly by associating values with ports. It is possible to specify finite state machines by defining a cyclic connection of components, using ports to encode the state variables. However, this does not permit describing the behavior of components that have internal state, i.e. state that is in addition to that of their ports. The example below illustrates associating internal state with a D Flip-Flop D1:

```
(if (and (true (value (port clock D1) high) $t)
         (true (value (port clock D1) low) (1- $t))
         (true (value (port data  D1) $v) $t))
    (and (true (value (state contents D1) $v) $t)
         (true (value (port output D1) $v) $t)))
```

This rule states that whenever there is a rising edge at the clock input both the output of D1, and its local state variable *contents*, have the same value as that of the *data* input.

One can specify an arbitrary number of local state variables, where each state variable is a term of the form:

```
(state <variable-name> <device>)
```

Additional examples of behavior specification can be found in Appendix A, which gives the complete definition for D74.

2.3. Specifying Structure and Behavior Hierarchically

Up to this point we have only given examples of non-hierarchical design specifications. One of the primary goals of MARS is to be independent of any design level or methodology. This is made possible by allowing the user to define the vocabulary and behavior of the primitive components. In addition to primitives, the user can also create prototypes which define a composite object. Prototypes are templates for instantiating a class of similar, or identical, components. Prototypes are the only way to define hierarchical boundaries in the specification of a design. The example below defines the sub-structure of a prototype called d74.

```
(prototype d74
   ((subpart* d74 m1 m2 m3 a1 a2)
    (type m1 multiplier) (type m2 multiplier) (type m3 multiplier)
    (type a1 adder) (type a2 adder)
    (conn  (port out m1) (port in1 a1))
    (conn* (port out m2) (port in2 a1) (port in1 a2))
    (conn  (port out m3) (port in2 a2))
    (subconn* (port in1 d74) (port in2 m1) (port in1 m2))
    (subconn* (port in2 d74) (port in1 m1) (port in2 m3))
    (subconn* (port in3 d74) (port in2 m2) (port in1 m3))
    (subconn  (port out a1) (port out1 d74))
    (subconn  (port out a2) (port out2 d74))))
```

The structural specification of a prototype must include information about the subparts, their types, the connections within the hierarchy level, and the inter-hierarchy connections. It is important to be able to distinguish between these two types of connections to identify the top-level and sub-structure behavioral specifications independently. The *subconn* and the *subconn** relations are used to define the inter-hierarchy connections.

There are two types of hierarchies: the structural hierarchy and the behavioral hierarchy. The structural hierarchy is used to encapsulate a collection of components at the same abstraction level. The previous example illustrates the structural hierarchy where there are no transformations in the data types of the signal across the hierarchy boundary—all signals are at the integer level. The structural hierarchy is used to capture more complex behavior by functionaly composing the behavior of simpler components at the same level. Similarly, the behavioral hierarchy is used to translate information between

levels (an example of this can be seen later for the description of adders).

We can also define the behavior of D74 at the top level independent of its sub-structure.

```
(protobehavior d74
  ((if (and (true (value (port in1 d74) $a) $t)
            (true (value (port in2 d74) $b) $t)
            (true (value (port in3 d74) $c) $t)
            (= $d (+ (* $a $b) (* $a $c)))
            (= $e (+ (* $a $c) (* $b $c))))
       (and (true (value (port out1 d74) $d) $t)
            (true (value (port out2 d74) $e) $t)))))
```

This definition provides a declarative behavioral specification for every instances of D74. The general form for specifying the top level behavior of a prototype is:

```
(protobehavior <dev> (<rule1> ... <rulen>))
```

It is also possible to define parameterized prototypes, which can be used to instantiate a class of similar components. For example, one can parameterize the definition of adders to include a width field, which specifies the number of bits at each input to the adder.

```
(prototype (adder n)
  ((if (range $i 1 n) (subpart adder (num fa $i)))
   (if (range $i 1 n) (type (num fa $i) full-adder))
   (if (and (range $i 1 (1- n))
            (= $j (1+ $i)))
       (conn (port cout (num fa $i)) (port cin (num fa $j))))
   (if (and (range $i 1 n) (= $j (1- $i)))
       (if (and (true (value (port in1 adder) $v) $t)
                (= $b (bit $j $v)))
           (true (value (port in1 (num fa $i)) $b) $t)))
   (if (and (range $i 1 n) (= $j (1- $i)))
       (if (and (true (value (port in2 adder) $v) $t)
                (= $b (bit $j $v)))
           (true (value (port in2 (num fa $i)) $b) $t)))
   (if (and (true (value (port sum (num fa 1)) $a) $t)
            (true (value (port sum (num fa 2)) $b) $t)
            (true (value (port sum (num fa 3)) $c) $t)
            (true (value (port sum (num fa 4)) $d) $t)
            (= $e (integer $d $c $b $a)))
       (true (value (port out adder) $e) $t))))
```

This example makes use of the *range* predicate which is useful for specifying an iteration of relations. For example, the second "if" statement declares that `(num fa 1)` (the first fa

of d74) through **(num fa 4)** are all full-adders. The *bit* function is used to select a particular bit field from an integer, while the *integer* function defines the integer corresponding to a sequence of bits.

The definition of the adder prototype defines a behavioral hierarchy. The data-types of the signals flowing in and out of the adder are at the integer level, while those at the full-adders are at the boolean level. The last three rules in the prototype definition perform these transformations. These prototype definitions are converted into an internal form which are then used for simulation (see Appendix B).

In this example we have specified a parameterized definition for adders which takes a single argument. The statement **(type baz (adder 4))** specifies an adder composed of four full-adders. It is also possible to define parameterized prototypes that take an arbitrary number of arguments. For example, one could define an adder with two parameters: the number of bits in the data path and the delay. The specification of such a prototype might start off as:

```
(prototype (adder n d) ..... )
```

The variables **n** and **d** can be used inside the prototype definition to specialize any particular adder. In this case the variable **n** can be used, as before, in the *range* relation for defining and connecting the appropriate number of full-adders. Similarly, we can replace **$t** in the conclusion part of the rules in the previous definition with **(+ $t d)** to include the delay. It is possible to specify the behavior of parameterized prototypes at the top level. For example, Appendix A gives the complete description of D74, which includes the top-level behavior description for parameterized adders.

2.4. Extending the Vocabulary for Describing Behavior

In specifying the behavior of components declaratively we can either enumerate the input combinations, or use the algebra of the abstraction level to perform the simplifications. An example of enumeration is given below for the *and* gate primitive:

```
(if (and (type $x and)
         (true (value (port in1 $x) false) $t))
    (true (value (port out $x) false) $t))

(if (and (type $x and)
         (true (value (port in1 $x) true) $t)
         (true (value (port in2 $x) $v) $t))
    (true (value (port out $x) $v) $t))
```

Alternatively, if the system understood how to simplify boolean expressions we could define the behavior of the *and* gate primitive by:

```
(if (and (type $x and)
```

```
        (true (value (port in1 $x) $a) $t)
        (true (value (port in2 $x) $b) $t))
(true (value (port out $x) (and $a $b)) $t))
```

In creating a new abstraction level we must define the primitives and the algebra for manipulating symbols at this level. In addition, we must define the relations between symbols at this level and adjacent levels. For example, we may wish to create a *gate* level abstraction with *and*, *or*, and *not* primitives, using boolean algebra to manipulate symbols at this level. In addition we may wish to relate signals at this level to the integer level.

In defining a new vocabulary we must define: the set of symbols that form the domain of the new type; and the set of operations and their actions over this domain. For example, in boolean algebra the domain of signals is *true* and *false*, and a complete set of operations is *and*, *or*, and *not*.

The user must define the domain of the new data type, but this information need not be included in MARS. The system does not perform any type checking to ensure that all expressions are well formed. The actual type checking can be done by the user if the domain has been specified. The actual specification of the data type is done using the *dtype* relation. The general form of such a relation is **(dtype <value> <type>)**. For example, the domain specification for boolean algebra is given below:

```
(dtype true boolean)
(dtype false boolean)
```

The operations of a new data type are restricted to occur anywhere an expression can occur in a proposition. For example, in the **<value>**, and **<time>** field of propositions of the following form:

```
(true (value (port <port-name> <part-name>) <value>) <time>)
```

The terms **<value>** and **<time>** in the previous proposition refer to arbitrary expressions over a data type. Sometimes these expressions are members of the domain and stand for themselves, for example the term **false** in **(true (value (port in bar) false) (+ 2 1))**. In other instances the algebra of the domain must be applied to simplify these expressions, as is the case with the term **(+ 2 1)**.

A *symbol* forming the expression field of a proposition stands for itself and is taken literally (including variables). Any non-atomic expression is assumed to be in prefix form. The general form of a non-atomic expression is **(<operation> <arg1> ... <argn>)**, where the arguments themselves can be arbitrary expressions. If MARS is not informed that the **<operation>** is an operator the expression itself is taken literally.

To declare that a certain symbol is an operator we must use the *opr* relation. The general form of this relation is **(opr <function-name>)**. For example, we can declare the following operations for boolean algebra:

```
(opr and)
```

```
(opr or)
(opr not)
```

This scheme does not permit operator overloading— each operator symbol must be unique. For example, it is not possible to have two "+" operators: one for addition and another for set union.

The actual simplification of boolean expressions is defined via substitution rules which specify what an expression is equal to. The following propositions define the base level axioms for simplifying boolean expressions without variables:

```
(= true true)
(= false false)
```

The primitive data types at the boolean level are equivalent to themselves. We can now extend the definition to include the *not* operator:

```
(if (= false $x) (= true (not $x)))
(if (= true $x) (= false (not $x)))
```

The negation of an expression which reduces to *false* is *true*, and vice versa. The simplification rules for the *or* operator are given below:

```
(if (= true $x) (= true (or $x . $y)))
(= false (or))
(if (and (= false $x) (= $v (or . $y))) (= $v (or $x . $y)))
```

The first rule states that if the first term reduces to *true*, the entire expression can be replaced with *true*. The last rule forces the remaining terms to be checked if the first term is *false*. Similarly, we can define the simplification rules for the *and* operator:

```
(= true (and))
(if (and (= true $x) (= $v (and . $y))) (= $v (and $x . $y)))
(if (= false $x) (= false (and $x . $y)))
```

2.4.1. What is Provided

The system does not provide any pre-defined abstraction levels, or primitives, for describing a design. However, it does have a limited knowledge of arithmetic by understanding the addition (+), subtraction (−), multiplication (•), division (//), increment (1+), and decrement (1−) operators. In addition, it understands the relational operators for comparing integers (<, <=, >, >=).

The system also provides two operators for translating information between the integer and boolean abstraction levels. The *bit* operator is used to select a particular bit field from a positive integer. The general form for such an expression is **(bit <bit-number> <integer>)**. The least significant bit corresponds to a bit-number of 0. Similarly, the *integer* operator is used to define the positive integer corresponding to a sequence of bits.

The general form of this operator is **(integer <bitn> ... <bit0>)**. The arguments from left to right are from the most significant to the least significant. Examples of the use of these operators can be found in section 2.3 for the specification of the adder prototype.

The primitives provided by MARS are inadequate for direct use in most applications. MARS is best viewed as a framework for implementing discrete event driven simulators. An implementer of a simulation tool can tailor MARS to a particular domain by specifying a set of abstraction levels. The actual creation of abstraction levels is facilitated by the flexibility of creating new primitives and vocabularies. The primitives can either be atomic, or composite if they are defined via prototypes. The declarative specification of behavior can either enumerate the input combinations, or use the algebra of the abstraction level to generate the behavior.

An end user of a specialized instance of MARS need not be concerned with the definitions of the primitives, or the definition of the vocabulary. An implementer can also define the prototypes for translating a component at one abstraction level into a collection of components at the next lower level. If a user is not permited to define such prototypes we are left with an extremely simple design refinement system. The user need only specify the structure of the design at the highest abstraction level, since there is only one pre-defined way to realize a component at the next lower abstraction level.

In general, design is not a simple top down process. The art of designing involves trading off between alternate ways of implementing a component at lower abstraction levels. Information is often passed in both directions across hierarchy boundaries, and earlier decisions may need to be revised. In such an environment it is essential to permit an end user to define prototypes for translating between abstraction levels. However, the vocabulary of the various abstraction levels can be predefined.

A more radical approach would be to provide an end user with raw MARS, and let him define both the abstraction levels and the design. This scheme has the greatest flexibility by not forcing a designer to work in a rigid hierarchy of abstraction levels, at the expense of greater effort. However, it might be possible to reduce this effort by providing a library of abstraction levels for all designers to use.

3. Reasoning with Design Descriptions for Simulation

The task of the simulator is to take the description of the structure and behavior of a design, and the stimulus at the inputs, as data in generating its behavior. MARS provides an environment for describing and simulating devices whose behavior is expressed discretely. For certain abstraction levels there is a well defined algebra and set of procedures for describing and reasoning about continuous behavior. For example, the behavioral description of *n*MOS gates in terms of a set of differential equations, and the calculus for solving these equations. MARS is not suited for reasoning about behavior at such abstraction levels, due to the inefficiency of simulating continuous behavior by a sequence of discrete steps.

As mentioned earlier, MARS does not assume a particular unit of time for simulation. It is up to the user to ensure that all behavior rules are mutually consistent by using the same unit of time (nano-seconds, etc.). The actual times for which events are processed in a simulation are dependent on the device being simulated, and its inputs. That is, the simulation times are not a predefined sequence, say 1, 2, 3, into which all events are compartmentalized. The actual event times are directly computed from the behavioral rules of the components at simulation time. A particular simulation may have the following times at which events are processed: 1, 1.4, 1.5, 10, 11, 11.1, 1000, etc.

At each event time there are some ports whose value is changed. Therefore, the state of the device being simulated needs to be recomputed at every event time. For most complex systems, a change in a sub-component will only produce a local change in the state of the system. There are two basic control strategies that can be employed to recompute the state of the design at every event time: recompute the state for every component; or propagate the changes at a port to all other ports that depend on it.

To increase the efficiency of simulation, MARS uses an event driven control strategy, which propagates the changes in port values instead of recomputing the state of every component for each simulation time. The event-driven control strategy has the advantage of only recomputing the state of those ports whose value can be changed, at the expense of keeping track of this dependency information. This is specially advantageous when simulating complex systems where the affect of a change in a port value is local— to compute the state of every component for every time step would be prohibitively expensive.

3.1. Inference for Generating Behavior

The basic inference technique used by MARS for performing event-driven simulation is *forward chaining* on an explicit rule base. The examples given in the previous section for the description of behavior via rules are the data that is used by inference procedures. There is an isomorphic explicit representation of this design description within the computer— the information is not compiled into specialized data structures and pointers. The propositional representation methods and the *forward chaining* inference technique are provided by MRS.

For simulation, there are two important operations on the knowledge base provided by

MRS: looking-up, and asserting facts. The lookup operation is used to see if there are facts matching the query[1] in the knowledge base. If there is more than one fact matching the query, they are returned in the inverse order of their storage (most recently stored first). Asserting a fact invokes the forward chaining inference mechanism to perform the event driven simulation. When a fact is asserted it is stored directly in the knowledge base, and all facts that are logical consequences are also asserted.

To understand the operation of the forward chaining inference mechanism consider the following rule which describes part of the behavior of an inverter Inv1.

```
(if (true (value (port in Inv1) on) $t)
    (true (value (port out Inv1) off) $t))
```

When a fact is asserted, all rules that might depend on the fact are checked. The rules are checked in the inverse order of their storage in the knowledge base. A rule depends on a fact if the fact matches some precondition of the rule. When a rule is checked, the remaining preconditions of the rule are checked in sequence from the begining. If we assert the following fact, the previous rule is checked.

```
(true (value (port in Inv1) on) 3.4)
```

The precondition of the previous rule is satisfied by binding the variable `$t` to 3.4. Since there are no more preconditions in the rule the conclusion of the rule is asserted after substitution with the same variable bindings:

```
(true (value (port out Inv1) off) 3.4)
```

For a more interesting case consider a knowledge base with the following facts:

```
(if (and (type $x multiplier)
         (true (value (port in1 $x) $a) $t)
         (true (value (port in2 $x) $b) $t))
    (true (value (port out $x) (* $a $b)) $t))

(type foo multiplier)

(true (value (port in2 foo) 3) 4)
```

The first rule is the generic behavioral description for multipliers. The remaining facts state that `foo` is a multiplier whose first input port has a value 3 at time 4. Consider asserting the following fact:

```
(true (value (port in1 foo) 2) 4)
```

This fact matches the second precondition of the rule by binding the variable `$x` to `foo`, `$a` to 2 and `$t` to 4. In order to see if the entire rule is applicable the remaining preconditions are looked-up in sequence, with the above bindings. First the proposition

[1] More than one fact can match due to variables in the query or facts.

`(type foo multiplier)` is checked, which is stored directly in the knowledge base. The third precondition `(true (value (port in2 foo) $b) 4)` is checked next. This fact can be deduced from the data base by binding the variable `$b` to 3. All the preconditions of the rule are now satisfied. The variable bindings for the preconditions are then plugged into the conclusion to get:

```
(true (value (port out foo) 6) 4)
```

3.1.1. Simplifying Expressions

In the previous example the system had to simplify the expression `(* 2 3)`, and replace it with 6. The simplification of expressions is performed using the axioms that define the algebra of the vocabulary. The actual simplification of expressions is performed by using the *backward chaining* inference method [7].

The backward-chaining inference method is similar to forward-chaining, except that instead of finding all logical consequences of a fact we are interested in checking if a fact is the logical implication of all the facts in the knowledge base. To achieve a goal the inference method works backwards from the conclusions of a rule and tries to satisfy the preconditions. The inference method returns the bindings for the variables in the goal that make it the logical consequence of the facts in the knowledge base.

In simplifying an expression *exp*, MARS tries to deduce what the expression is equal to. The first deduction to provide an answer is assumed to be the simplification of the expression. For example, in simplifying `(and (or false true) false)` MARS uses the backward-chaining inference mechanism to prove the proposition `(= $$ (and (or false true) false))`. The inference method binds the value of the variable `$$` to an expression, which when substituted for `$$` in the previous proposition makes the entire proposition a logical consequence of the facts in the knowledge base. The actual answer returned is the binding of this variable, which represents the simplification of the original expression.

To illustrate the simplification of this expression, using the backward chaining inference method, consider the axioms for boolean algebra presented earlier in section 2.4.[2]

```
(= true true)
(= false false)

(if (= false $a) (= true (not $a)))
(if (= true $b) (= false (not $b)))

(if (= true $c) (= true (or $c . $d)))
(= false (or))
(if (and (= false $e) (= $g (or . $f))) (= $g (or $e . $f)))
```

[2] To simplify the explanations, the variable names have been made unique across all the rules— this is not required by MRS.

```
(= true (and))
(if (and (= true $h) (= $j (and . $i))) (= $j (and $h . $i)))
(if (= false $k) (= false (and $k . $l)))
```

The dot notation is used to take apart a structure by binding the remaining arguments in a pattern (if any) to a variable. For example, in matching `(or false true false)` with `(or $x . $y)`, the variable `$x` is bound to `true` and `$y` is bound to `(true false)`.

This expression cannot be simplified in one step, since there is no fact in the knowledge base matching:

```
(= $$ (and (or false true) false))
```

The only way to proceed is to check if the above fact matches the conclusion of some rule. In this case, it matches the conclusion of the second *and* rule: [3]

```
(if (and (= true $h) (= $j (and . $i))) (= $j (and $h . $i)))
```

by binding the variable `$$` to `$j`, `$h` to `(or false true)`, and `$i` to `(false)`. Before we can return an answer we must make sure the preconditions of the rule are also satisfied with these variable bindings. In this case we must prove `(= true (or false true))` and `(= $j (and false))`. In order to prove the first precondition we must use the third *or* rule:

```
(if (and (= false $e) (= $g (or . $f))) (= $g (or $e . $f)))
```

by binding the variable `$g` to `true`, `$e` to `false`, and `$f` to `(true)`. We now must satisfy the preconditions of this rule, with these bindings added to the previous bindings, to prove `(= false false)` and `(= true (or true))`. The first precondition is directly stored in the data base, and the second precondition can be proved by application of the first *or* rule:

```
(if (= true $c) (= true (or $c . $d)))
```

We now must go back and prove the second precondition of the first rule that was matched, i.e. prove `(= $j (and false))` with the above variable bindings (the variable `$j` was not further bound in proving the first precondition). This fact can be proved by using the third *and* rule:

```
(if (= false $k) (= false (and $k . $l)))
```

by binding the variable `$j` to `false`. The answer to return is the binding of the variable `$$`. However, this variable is itself bound to `$j`. Therefore, the answer to return is the binding of the variable `$j`, which is `false`.

This section has illustrated simple examples of the the forward chaining and backward chaining inference methods. The operation of these inference methods is tied intimately to that of the *most general unifier*, or MGU, in logic [7]. When a rule is checked, each precondition can be satisfied in a number of ways— each for a particular binding of the

[3] It also matches the third *and* rule, but the preconditions of that rule cannot be satisfied.

variables in that precondition. It is the task of the inference mechanism to find all consistent cross-products of bindings for the preconditions. A collection of variable binding is inconsistent if the same variable is bound to two or more different expressions. For forward chaining, each consistent set of variable bindings for the pre-conditions of the rule are plugged into the conclusions, and asserted. Similarly, for the backward chaining, all consistent bindings that prove the preconditions must be returned.

3.2. Partitioning Simulation State Information

The previous examples have illustrated that the forward chaining inference mechanism is adequate for performing event driven simulation for devices with zero delay. The forward chaining inference method of MRS performs a depth first propagation of the consequences of a fact that is asserted— it has no notion of time. However, MARS permits devices to have arbitrary delay, and the pure forward chaining inference mechanism cannot separate facts that are to be true at some time in the future from those that must be true at present.

At any point, the simulator is processing the state of the device for a particular time step. All facts that are asserted for the current time must be propagated, similar to forward chaining, immediately. If a fact is asserted to be true at some point in the future (greater than the current simulation time) it must be saved, and asserted later at the appropriate time. For example, consider the multiplier described earlier, but this time a delay of three time units from its inputs to the output:

```
(if (and (type $x multiplier)
         (true (value (port in1 $x) $a) $t)
         (true (value (port in2 $x) $b) $t))
    (true (value (port out $x) (* $a $b)) (+ $t 3)))

(type m1 multiplier)

(true (value (port in2 m1) 3) 4)
```

If we now assert the fact `(true (value (port in1 m1) 2) 4)` at time 4, the forward chaining inference mechanism will deduce `(true (value (port out m1) 6) 7)`. If we assert this fact immediately there is the possibility of making incorrect deductions. If `m1` is connected to another device, as in d74 where it is connected to the first input of the adder `a1`, asserting `(true (value (port out m1) 6) 7)` at time 4 will cause the behavior rules of the adder `a1` to be checked at time 4. The problem is that at time 4 there may not be enough information to predict what the second input to the adder `a1` will be at time 7 — we can only use the current value. The solution is to store all facts that are to be true in the future in an event queue, and process all facts about the current simulation time immediately.

The meta-level capabilities of MRS are used to select the appropriate action when asserting a fact— either forward chaining or storing it for later retrieval. The event queue

is a balanced heap [1] of simulation times. When a fact is to made true in the future, the time at which it is to be made true is entered in the heap, and the fact is associated with this time. After processing all the events for a given time step the simulator retrieves the next simulation time by removing the smallest element from the event queue heap. The processing of the next time step starts off by asserting all the facts that were associated with it. By representing the event queue as a balanced heap the insertion, and deletion, of simulation times is reduced to $O(\log n)$ (n is the number of event times in the queue).

3.3. Managing the Frame Problem

The event queue mechanism partitions the facts about port values into two classes: those that are true at the current simulation time, and those that must be true in the future. This is sufficient for performing event driven simulation for devices with arbitrary delays, as long as the behavior rules do not refer to any past simulation state. In MARS we would like to provide the flexibility of permitting the behavior rules to refer to the state of the simulation at arbitrary times in the past. An example of this was seen on page 6 for the description of the D Flip-Flop. The detection of the rising edge of the clock was based on knowing that the clock is high at the current time, and that it was low one time unit in the past.

To permit this flexibility, MARS provides the ability to store the entire simulation state, from the start of the simulation. It is possible to record the entire state of the design for every simulation time by separately storing the state of the design for every time step. However, this would be extremely inefficient in the utilization of storage. The motivation for the event driven control strategy was to minimize the processing of information, since only a small part of the design is expected to be active at any time step. The same observation can be used to realize that only a small subset of the port values should change at any time step. Therefore, most of the design state information for each simulation time will duplicate information stored for earlier times.

Instead of storing the entire state of the design for every time step, we can store only those facts that have been made true at a simulation time, with that time. This mechanism only records the change of state at each simulation time. However, there is an added complication with this representation scheme. Earlier, in checking the pre-conditions of rules we only had to lookup the facts in a single knowledge-base. By partitioning the simulation state, we must specialize the lookup function to extract the information from the appropriate partition.

Figure 2 illustrates the partitioning of information for a simple circuit consisting of an inverter and a buffer, modelled as unit delay elements. The simulation was started at time zero with both nodes **a** and **b** set to zero, and has progressed to time three. The boxes in the figure partition the information in the knowledge base. The box labelled 0 contains those facts that were made true at time 0, and so on.[4]

[4]The actual syntax has been simplified, for example, a=0 is a shorthand for (value (port a inv) 0). Also, the port values of the

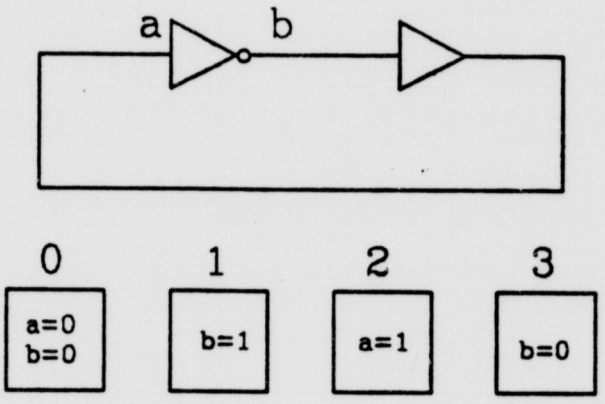

Figure 2: Partitioning the Simulation State Information

If we want to check if a fact is true at a time t, we must start searching for the fact from the largest partition less than, or equal to, t. If the fact is not found to be true in this partition we must search through the previous simulation time partitions sequentially. For example, suppose we wanted to check if `(true (value (port b inv) $x) 2.5)`, that is, what value port b of the inverter has at time 2.5. The largest simulation time less than, or equal to, 2.5 is 2, and there is no information stored in this partition for port b. The previous simulation time step (time 1), however, includes a fact that states that port b has a value 1.

Even with the partitioning of information, to store only the changes in simulation state, the amount of information to store can become unmanageable. To cope with this MARS permits the user to select the amount of history to save in a simulation. To use the least amount of space the user can select to keep only a single value for a port. Whenever a new value is stored for a port in the current simulation partition, the previous value (if any) is deleted (from whatever partition it was in).

An alternative is to save information up to n time units in the past. As the simulator proceeds from one time step to the next it deletes all facts from those partitions that are more than n time units in the past from the *next* simulation time. There is an exception to this rule to prevent all information about a port to be deleted. If a port value is not changed for a long time, its latest state will be recorded in a partition which is to be garbage collected. If all facts in this partition are deleted we will lose all information about the state of this port. To prevent this, only those facts that record duplicate (earlier) values of a port are garbage collected from garbage collectible partitions.

---

buffer have been left out.

3.4. Maintaining Dependencies

In simulating devices with zero delay, the pure depth first forward chaining inference mechanism can produce erroneous results. An example of this can be seen for the circuit in Figure 3, which shows an *and* gate and an inverter, both of which have zero delay. The logic equation for the output of this circuit is $a \wedge \neg a$, which simplifies to *false*. Since MARS permits describing such structures it is essential that it simulate them correctly.[5]

The problem with the depth first processing of events is that there is no pre-defined sequence of traversal at fanout nodes that is guaranteed to give the correct results in all situations. In the circuit of Figure 3 we can see a fanout leading to the input of the inverter and the second input of the *and* gate. If we process the events along the inverter branch first, the output will have a $0 \rightarrow 1 \rightarrow 0$ glitch when there is a falling edge at the input. The falling edge at the input will first be propagated to the input of the inverter, setting its output to one. This will cause the behavior rules of the *and* gate to be checked, which will further check the value at the second input. Since the change at the input has not been propagated to the second input of the *and* gate, the old value 1 will be found. Only after the output of the *and* gate has been set to one will the falling edge at the input be propagated to the second input of the gate. This will cause the output to be set to the correct value of 0, but only after it was set incorrectly to 1. A similar situation exists if we chose the other order at the fanout node, and there is a rising edge at the input.

It is not possible to solve this problem by dynamicaly altering the control of the inference mechanism based on the dependencies of port values. One port depends on another if the latter can affect its value. The selection of events based on dependencies is not possible, in general, because there is no partial order dependency relation for a cyclic connection of components.[6] In order to select the correct event for simulation, we are forced to perform a simulation. This scheme has the added disadvantage of sorting all events dynamicaly at simulation time.

In MARS we have solved this problem by maintaining the justifications for each fact. The justifications record all facts in the premise of a rule with the conclusion when it is asserted. If any premise is removed, all facts that depended on this premise are also removed. An advantage of using the dependency mechanism is that the justifications can be used to provide an explanation facility for the actions of the simulator.

If a node is asserted to have a certain value at time t, and a different value already exists at time t, the old value is removed. There is an implicit assumption that the latest information is more correct than what was known before. This assumption is based on the fact that the old values represent the state of a port in the past and the new value represents the correct value for the current simulation time. Figure 4 illustrates the dependency relations for the previous example. The arrows depict the actual dependency relation— a fact at the tail of an arrow depends on all facts at the heads of arrows emanating from it.

---

[5] Independent of whether they are physically realizable.

[6] It is possible to order the events if zero delay cycles are prohibited. This would preclude the simulation of self-timed timed finite state machines where there are no clocks to break the feed-back paths.

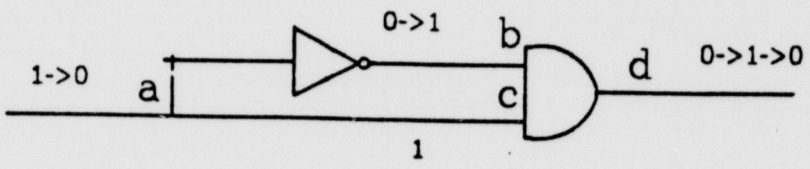

Figure 3: Simulation of zero Delay Components

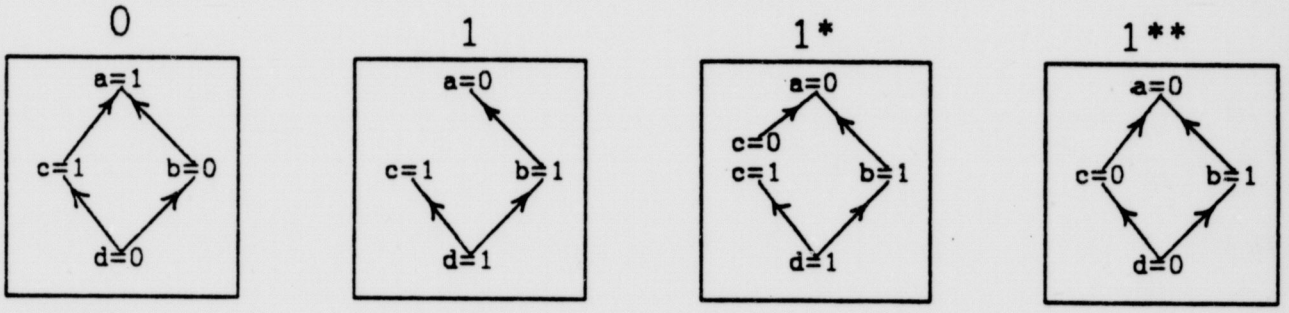

Figure 4: Maintaining Dependency Relations

Initially the circuit is in steady state at time 0, with all ports having the values shown in partition 0. Let the value of the input a change to 0 at time 1. The consequences of this change are first propagated along the inverter branch, as shown in partition 1. When the value of the port c is checked, the old value 1 is found from partition 0, which causes port d to be set to the incorrect value 1. The processing for time 1 is not complete, however, since the change at port a needs to be propagated along the second fanout branch to port c. This is illustrated in partition 1* [7] where port c has both a value of 0 and 1. The old value of 1 is deleted, and in addition, all facts that depended on port c being 1 are deleted— in this case, the incorrect value for port d. The final state for simulation time 1 is shown in the partition 1**.

In simulating combinational circuits it is not necessary to record the dependency information. When a new port value is asserted, the old value can simply be overwritten, and the consequences of the new value propagated in a depth first manner. The simple overwriting of port values is not adequate when simulating devices with state. An erroneous

[7] Which is a later snapshot of the state of the partition for simulation time 1.

input can take a finite state machine from state $S_i \rightarrow S_j$. When the correct input is later applied, it isn't necessary that the outputs, or the next state, in state S_i be the same as those in state S_j.

4. Providing a Flexible Simulation Environment

The declarative specification of the structure and behavior can be used to provide a flexible simulation environment. We can take advantage of the hierarchical specification of the structure of a design to perform mixed-mode simulation, with different parts of a design simulated at different abstraction levels. It is also possible to verify a design by comparing the top-level behavior with that of its implementation. We can take advantage of the ability to create a new vocabulary within MARS to perform symbolic simulation. Instead of propagating actual values through a design we can propagate symbols (variables) that stand for an entire class of values. The same scheme can be used for performing simulation with partial information.

These ideas only represent some of the ways of utilizing the declarative specification of a design, in conjunction with AI techniques, for providing a flexible simulation environment. There are many more ways of exploiting the declarative specification of a design by applying AI techniques, however, the remainder of this section will only concentrate on these.

4.1. Selection of the Simulation Level

The actual selection of the simulation level is done via the *simsub*, and *simtop* relations. The general form of these relations is **(simsub <part-name>)**, or **(simtop <part-name>)**. The **<part-name>** corresponds to the usual hierarchical naming scheme for a component, and can include variables in arbitrary positions.

The *simsub* relation declares that a part is to be simulated in terms of its sub-components, and the *simtop* relation indicates that the top-level behavior is to be used. For example, **(simsub (part $x d74))** indicates that all sub-parts of d74 are to be simulated in terms of their sub-structure. A simsub relation for a primitive, or a *simtop* relation for a composite without top-level behavior, does not generate any behavior.

A component is not simulated at all if it has neither a *simsub* or *simtop* relation. This is quite useful for eliminating parts of a design from being simulated.

It is possible to simulate a component at both the top-level and the sub-structure if it has both a *simsub* and *simtop* relation. This is useful for providing a simple verification scheme. When a new prototype is created we can test its definition by simulating it at both levels, and compare the results. Once the verification is complete, all instances of the prototype need only be simulated at the top level. This verification scheme is limited by the completeness of the test vectors, and does not guarantee correct behavior in all situations.

When a component is simulated at both levels, a demon is created that compares the values assigned to the boundary of the hierarchy from the top level and the substructure. If at the end of a time step these values are different, an error message is reported.

4.1.1. Automatic Selection of Simulation Level

The user can set the simulation levels for the components manually, or allow the system to do this automatically. The automatic selection of simulation level will decide if a component is to simulated at all, and if it is, at what level it should be simulated. The automatic selection process will never select both the top level and the substructure of a component concurrently. If the user is interested in verifying a component (subject to the test vectors), it is up to him to ensure that the component has both a *simsub* and *simtop* relation.

For a given simulation, the user must specify what nodes he is interested in monitoring. The information of these traced nodes is used to automatically decide the appropriate simulation level for all components.

Initially the database does not include any facts about the simulation level of components. In the absence of this information, all components will be inactive during a simulation. To select the simulation level automatically the system reasons backwards through the *structure* of the design from the traced nodes.[8] The primitive propagation steps involve tracing back along the connections between components. Reasoning backward through a component causes the processing to be performed recursively for all inputs of the component. On encountering a component, in the backward reasoning process, the system sets it to be simulated at the top level. However, reaching a hierarchy boundary causes the super-component to be simulated at its sub-structure. Since it is possible to trace an output of a composite and a node in its sub-structure concurrently, the *simsub* relation has precedence over *simtop*.[9]

In addition to performing simulation at the highest level, the automatic selection of the simulation level has the added advantage of removing parts of a design from simulation if they cannot affect any of the nodes of interest.

4.2. Symbolic Simulation

In generating the behavior of components, MARS does not have any requirement that ports be assigned values from their signal domain. The system is only satisfying symbolic constraints between the ports of a design. We can take advantage of this to perform symbolic simulation by assigning expressions to the ports of a design. It is up to the behavior rules of the design to propagate the expressions in an intelligent manner.

In order to perform symbolic simulation we must alter the behavior rules of the components, or the algebra of the abstraction level, to propagate expressions at the input ports. Some of this is done automaticaly for us if the preconditions of the behavior rules do not consider all the inputs. For example, the first rule for the *and* gate, in section 2.4, states that the output is false if the first input is false. Even if the value of the second input is

---

[8] The actual propagation is done using the MRS *residue* planning mechanism.
[9] In this case, asserting (simsub <foo>), (simtop <foo>) is removed.

an expression, the output *will* be false. However, the behavior rules for the *and* gate do not consider the case when the first input is an expression, or both inputs are expressions. The following rules must be added for the *and* gate to propagate arbitrary expressions at the inputs:[10]

```
(if (and (type $x and)
         (true (value (port in2 $x) false) $t))
    (true (value (port out $x) false) $t))

(if (and (type $x and)
         (true (value (port in2 $x) true) $t)
         (true (value (port in1 $x) $v) $t))
    (true (value (port out $x) $v) $t))

(if (and (type $x and)
         (true (value (port in1 $x) $a) $t)
         (true (value (port in2 $x) $b) $t)
         (not (dtype $a boolean))
         (not (dtype $b boolean)))
    (true (value (port out $x) (and $a $b)) $t))
```

It is important to *separate* the algebra for performing the simplifications from the behavioral description of devices. If the simplification is done by the behavioral rules of a device, as above, similar simplifications must be done by all devices at the same abstraction level (*or*, *xor*, *nand*, etc). This redundancy obscures the description of a component by including information that has nothing to do with its normal behavior. It would be better to extend the algebra of the abstraction level so that the same algebra can be shared by all devices at that level. The following facts must be added to the declarative specification of the boolean operators in section 2.4 to simplify conjunctive expressions:

```
(if (dtype $x variable) (= $x $x))
(if (= false (and . $y)) (= false (and $x . $y)))
(if (and (= true (and . $y)) (= $v $x))
    (= $v (and $x . $y)))
(if (and (= $v $x) (not (dtype $v boolean))
         (= $w (and . $y)) (not (dtype $w boolean)))
    (= (and $v $w) (and $x . $y)))
```

The first rule states that all variables are equivalent to themselves, and cannot be further simplified.[11] The last rule applies when an expression cannot be reduced using the simpler techniques, that is, when both the head and tail of a conjunct simplify to symbolic expressions. The simplifier of section 2.4 was guaranteed to produce the simplest expressions

[10] There is an implicit assumption that any non-boolean value is a symbolic boolean expression.
[11] The system defines the data types of all variables to be `variable`.

when there were no variables. The addition of these rules only guarantees that the *correct* expressions will be propagated at the boolean level. These axioms do not capture all the simplification rules that apply in the presence of variables. For example, the distributive law, DeMorgan's law, etc. To perform symbolic simulation the user, at the very least, must extend the behavior of the devices, or the vocabulary, to propagate expressions. Additional simplification rules can be included to enhance the quality of the values propagated.

By similarly extending the simplifier for disjunctions, it is possible to perform symbolic simulation by assigning variables, instead of constants, to the inputs of a design. The outputs of a symbolic simulation will be expressions in the domain of the signal type. We can utilize this mechanism to perform ***simulation with partial information.*** Those inputs whose value is unknown can be assigned variables at the start of the simulation, and the other inputs are assigned their known values.

The task of symbolic simulation is complicated by components whose behavior involves conditional branching. In propagating an expression through such devices, we must simulate the design for all possible values of the conditional. At each conditional there are a number of possible branches to select propagating information on. In certain situations we must simulate each branch separately, for example, if the input expression to the conditional is a variable, which can have any value over the domain. However, it is possible to make a more intelligent decision, in other cases, and eliminate certain branches of the conditional from consideration. For example, at the integer level, if the input to the conditional is `(* 2 $v)`, we need not consider any branches that require an odd value for the conditional.

Each branch of a conditional requires a certain input condition to be satisfied, to be applicable. In simulating any branch of a conditional, we must constrain the input expression to satisfy the condition of the branch. This may require the binding of variables in the input expression, for example, if the condition for a branch is `false`, and the input expression is `(not $v)`, we must bind the variable `$v` to `true`. We must substitute this binding in all expressions involving the variable `$v` for this branch of the simulation.

5. The Utility of Hierarchy for Simulation

Most devices are functional, that is, given the inputs there is only one possible state for each output. Simulation, therefore, does not involve any search, as is the case with reasoning backwards through a component, for example, in automatic test generation. In spite of this, we can improve the efficiency of simulation dramaticly by taking advantage of the hierarchical specification of a design.

The hierarchical description of a design includes both the hierarchical specification of structure and behavior. Prototypes define hierarchy boundaries, and partition the structure of a design. By associating a behavioral description with a prototype we define the behavioral hierarchy of a design. To be useful, a hierarchical design specification must include the hierarchical specification of behavior. If a composite component does not have a top-level behavior description defined for it, it must be simulated at its sub-structure. However, if such a description is available, we have the choice of simulating the component at either the top-level, or in terms of its sub-components.

In general, it is advantageous to perform the simulation of a component at the highest abstraction level. By simulating at higher levels we reduce the number of components that the simulator must reason with. The actual reduction will be exponential in the number of levels of hierarchy that have been bypassed, by using a single high level behavioral description.

Although the behavior of a composite component is more complicated than that of the components at its sub-structure, it isn't the case that more complex behavior requires more complex processing of information to generate. For example, we can use tables to encode the output(s) of a component any given inputs. The higher level behavioral description of a component, using tables, caches the input/output relations that the lower levels in the hierarchy must compute. The actual savings depend on the function of the component—the savings factor can be greater than the number of components at the lower levels, since each component may need to be simulated a number of times, for designs with cyclic connections.

In certain situations there is a direct mapping between the higher level behavior of a component and instructions of the machine on which the simulator is running. We can take advantage of this by using the parallelism of the hardware to generate the behavior of a component. For example, we can use the multiply instruction of the machine to generate the behavior for a component that is a multiplier. The savings here are similar to those for tables, with the advantage that the storage space for the tables has been eliminated at the expense of a little computation.

The higher level description of behavior can also be simpler by abstracting away irrelevant details that are present at the lower levels in the hierarchy. For behavioral hierarchies, it is sometimes the case that a class of values at the lower abstraction level correspond to a single value at the higher level. For example, in translating information between the

signal and boolean levels, all voltage between 0 and 1 volt are equivalent to *false*. At the higher level the exact value is irrelevant, and computations that generate this detailed information are of no use.[12] A similar situation arises if we are only concerned with the sequence of events, and not the exact times at which the events occur. Simulating at lower levels in the hierarchy, where the exact time for events is calculated, will be an inefficient use of the computing resources.

The strict hierarchy of a design is often violated to satisfy the system constraints for speed, power, and area. Often the implementation of a design requires the sharing of components at lower levels to minimize the number of devices. For example, Figure 1 includes the design of a full-adder, where the first *xor* gate is shared between the logic for the sum and carry outputs. The actual sharing of components can lead to a redundancy of computation at the lower levels. For example, the output equation for the carry signal is:

```
carry-out = (or (and a b) (and a (not b) c) (and (not a) b c))
```

The simplest form of the carry output for the full-adder is:

```
carry-out = (or (and a b) (and a c) (and b c))
```

This equation is simpler, since there is no need for the higher level behavioral descriptions of the carry and sum functions to share any information. It would be impractical to automatically simplify the previous equation into the latter before generating the behavior, since this would require solving the boolean minimization problem, which is known to be exponential.

Another important advantage of simulating at higher levels in the hierarchy is that we can propagate a class of values with a single symbol. For example we can propagate a variable directly, as is the case with symbolic simulation. We can also propagate a variable with certain constraints, for example, the symbols *prime*, *odd*, *even*, *positive*, *perfect square*, etc. If the behavior we are interested in examining is at this level of detail, it is more efficient to propagate a single symbol instead of a set of symbols that it is equivalent to. For some of these symbols there is no analog at the lower abstraction levels, for example, there is no easy way to specify *prime*, or *perfect square*, at the boolean level— even with variables.

[12] Assuming that the lower level components have been verified to work correctly, that is, their outputs are never in an undefined state for any legal inputs.

6. User Interaction

MARS provides an interactive environment for hierarchically simulating a design. It has been successfully integrated with BAL [2] which provides an interactive environment for the *creation* of a design. These two systems form a part of the larger design entry workstation [6]. The interactive design environment aids the incremental development of a design by allowing the simulator to verify the operations of a sub-component of a design, as it is refined.

The ability to incrementaly verify the operations of a component, in the middle of the design process, greatly increases the utility of the design environment by allowing a designer to experiment with different design alternatives, and get immediate feedback to make the best decision. In addition, to enhance the simulation environment, MARS provides the ability to single-step the simulator, define break conditions, and trace the value of ports—capabilities normally found in software debugging environments.

The basic simulation cycle involves: creating a design description [13], or refining an existing design specification; specifying the initial inputs for simulation; selecting the nodes to be traced; defining the break conditions; and running the simulator for the appropriate time. On entering a break condition, or at the end of the simulation, the user can examine and/or change the state of the simulation at an arbitrary time in the past, present, or future.

Figure 5 shows a picture of a full-adder being simulated. The structure of the full-adder was specified graphicly, and the behavior was specified textually, using BAL. The same structural and behavioral descriptions are used for simulation. The menu on the right displays the command options available to the user, which can be selected using a mouse. For this simulation, the full-adder is being simulated in terms of its subcomponents, which are the two *and* gates, the two *xor* gates and the *or* gate. The initial inputs for simulation are:

```
(true (value (port cin foo) 1) 0)
(true (value (port cin foo) 0) 0)
(true (value (port cin foo) 1) 0)
```

that is, the carry-in and the first input of the full-adder are one, and the second input is zero. It is not necessary for the user to specify the values for all input ports at the start of the simulation. However, if port values are omitted, there is the possibility of not simulating parts of a design because the preconditions of rules cannot be satisfied.

Initially, the simulator does not select any nodes of a design to be traced. The user must select the nodes whose values he is interested in monitoring. For this simulation we have traced all nodes in the design. This was done using the ***add shade traces*** menu option, and selecting the entire full-adder using the mouse. All nodes start out in the undefined state, which is represented by a gray shade at the ports, for example, the carry output of the

[13] Not necessarily a complete description. Different parts of a design can be refined to different abstraction levels.

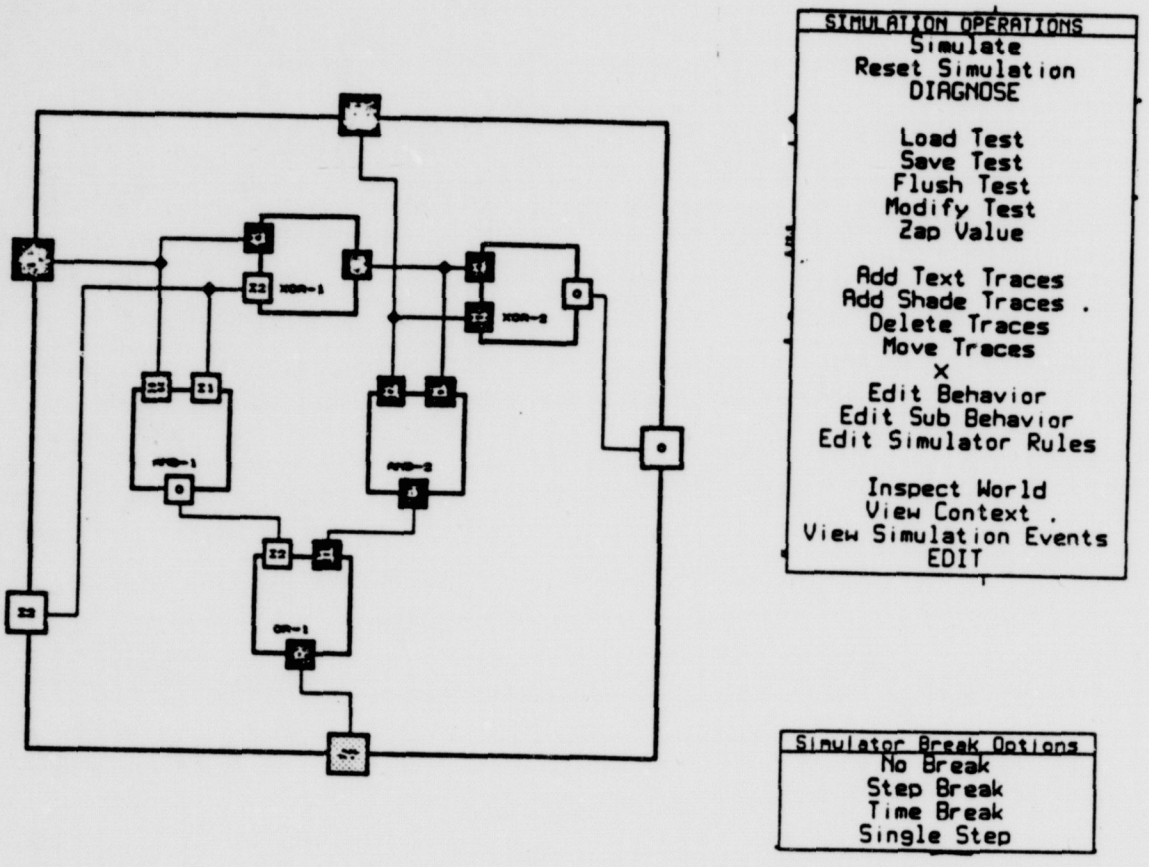

Figure 5: Simulation of a Full-Adder

full-adder in Figure 5. A value of *true* is represented by shading a port black, and *false* is represented by shading the port white. In addition to shade tracing, we can display the actual values being assigned ports.

During simulation, a change in the value of a traced port is displayed immediately next to, or inside, its graphical description on the screen. For example, as soon as the sum output changed to *false* the color of the output port was set to white. This graphical animation capability provides a richer environment for visualizing the flow of information through a design.

The user can also define arbitrary break conditions for a simulation. If any of the break conditions are satisfied during the simulation of a time step, the simulator enters a break after processing all the events for the given time step, and indicates which break condition was satisfied. The break conditions can be used to detect illegal inputs/outputs from a design, or to interrupt and examine the state of the simulation. For example, we may be interested in monitoring the inputs of the adder A1 of d74 to ensure that they are in the

proper range (not greater than 15— the maximum number that can be encoded by 4 bits). This can be done by defining the following break condition:

```
(and (true (value (port in1 a1) $x) $t) (> $x 15))
```

In addition to break conditions, the user can interrupt the simulation by specifying the number of event times to process before the next break; or the actual simulation time at which to enter a break, etc. The smaller menu, on the bottom right, in Figure 5 shows the options available to a user in selecting the simulation runtime and/or the number of simulation steps to process. On entering a break, the state of the design can be examined and/or modified, and the simulation continued.

6.1. Providing Explanations

MARS records the justifications for all facts in order to implement the truth-maintenance system described in section 3.4. The dependency information can be used to provide a simple explanation facility to justify the conclusions of the simulator. The user can ask for the immediate justifications for a proposition (up to one inference step). By selectively repeating this process, it is possible to trace the inference steps of the simulator in in reverse chronological order.

The system records the justifications for all propositions— both atomic and logical. The justifications include: the inference method used; the rule used by this inference method; and the facts that made the preconditions of the rule true. The user can inquire about the justifications for a fact by using the **why** command.[14] The example below is for the full-adder of Figure 5, with all port values defined at the end of the simulation.

To ask for the justification for the carry output bit being 1 at time 0 the user can select the **why** command and ask:

```
(true (value (port co fa) 1) 0)
```

The system will respond with:

```
P542: (TRUE (VALUE (PORT CO FA) 1) 0) by:
      SFA-ASSERT
      P658: (IF (TRUE (VALUE (PORT OUT O1) 1) 0)
                (TRUE (VALUE (PORT CO FA) 1) 0))
      P541: (TRUE (VALUE (PORT OUT O1) 1) 0)
```

This justification states that the carry output of the full-adder is 1 because the output of the *or* gate is 1. The symbols in front of the propositions are their unique names, and can be used to reduce the typing in asking questions. To get further justifications for the output of the full-adder being 1 we can ask **(why 'P541)**, and so on.

---

[14] This is a Lisp function, and not a relation symbol for propositions.

7. Conclusion

In building MARS we were investigating the appropriateness of applying AI tools to the task of simulation. We feel that the result of this experiment has been successful with the integration of this tool with the SUNDEW [6], and Palladio [3] design environments. These systems have to reason with designs described at a variety of abstraction levels, and require an extremely flexible simulation environment. The flexibility of MARS has permited its application to the simulation of domains other than digital circuits, including the simulation of a gasoline engine.

MARS provides a flexible device independent representation method for specifying a design. The user can not only specify the structure of a design, but can also create new abstraction levels by defining the behavior for the primitives. The vocabulary of MARS can also be extended to understand/simplify symbolic expressions at the new abstraction levels.

The incorporation of AI techniques is directly responsible for providing the capabilities to: perform symbolic simulation; perform simulation with partial information; automaticaly select the simulation level; verify the correctness of a design; and provide explanations for the conclusions of the simulator. In addition, the system also provides the debugging capabilities usually found in a software environment to help diagnose a design manually. The application of off-the-shelf AI techniques has also aided in the rapid implementation of the system.

MARS is a prototype system, with considerable room for improvement. The efficiency of the system can be improved by at least one order of magnitude by compiling out the meta-level axioms of the simulator. In addition, the capabilities of the system can be expanded by exploiting the explicit representation of the design and the AI tools provided by MRS. For example, we can enhance the explanation facility by providing justifications tailored to the understanding of the user. Also, it would be useful to automatically compile the behavior rules of a component into procedures incrementally, as its functionality is verified.

Acknowledgments

I would like to thank Harold Brown for his help in discussing the ideas that led to the development of MARS. I would also like to thank my advisor Mike Genesereth for his many comments that helped improve the organization and content of this paper. In addition, Harold Brown, Gordon Foyster and Hector Levesque also provided useful comments on drafts of this paper.

A. Design Description for D74

This appendix provides the definitions for the structure and behavior of D74. The definition includes the parameterized definitions for full-adders, adders, and d74s. The multipliers have not been further refined. These definitions instantiate the behavior rules for each gate.

```
Giving connections zero delay unidirectional behavior.

(if (conn $x $y)
    (if (true (value $x $v) $t) (true (value $y $v) $t)))


Description of "or" gates.

(if (type $x or)
    (if  (true (value (port in1 $x) 1) $t)
         (true (value (port out $x) 1) $t)))

(if (type $x or)
    (if  (and (true (value (port in1 $x) 0) $t)
              (true (value (port in2 $x) $v) $t))
         (true (value (port out $x) $v) $t)))

Description of "and" gates.

(if (type $x and)
    (if (true (value (port in1 $x) 0) $t)
        (true (value (port out $x) 0) $t)))

(if (type $x and)
    (if (and (true (value (port in1 $x) 1) $t)
     (true (value (port in2 $x) $v) $t))
        (true (value (port out $x) $v) $t)))

Description of "exclusive-or" gates

(if (type $x xor)
    (if (and (true (value (port in1 $x) 0) $t)
             (true (value (port in2 $x) 0) $t))
        (true (value (port out $x) 0) $t)))
```

```
(if (type $x xor)
    (if (and (true (value (port in1 $x) 1) $t)
             (true (value (port in2 $x) 0) $t))
        (true (value (port out $x) 1) $t)))

(if (type $x xor)
    (if (and (true (value (port in1 $x) 0) $t)
             (true (value (port in2 $x) 1) $t))
        (true (value (port out $x) 1) $t)))

(if (type $x xor)
    (if (and (true (value (port in1 $x) 1) $t)
             (true (value (port in2 $x) 1) $t))
        (true (value (port out $x) 0) $t)))
```

Behavioral description of multipliers.

```
(if (type $x multiplier)
    (if (and (true (value (port in1 $x) $y) $t)
             (true (value (port in2 $x) $z) $t))
        (true (value (port out $x) (* $y $z)) $t)))
```

Prototype Definition for full adders.

```
(prototype full-adder
           ((subpart* full-adder xor1 xor2 or1 and1 and2)
            (type xor1 xor)
            (type xor2 xor)
            (type or1 or)
            (type and1 and)
            (type and2 and)
            (conn* (port out xor1) (port in1 xor2) (port in2 and2))
            (conn (port out and1) (port in2 or1))
            (conn (port out and2) (port in1 or1))
            (subconn* (port in1 full-adder) (port in1 xor1)
                                            (port in1 and1))
            (subconn* (port in2 full-adder) (port in2 xor1)
                                            (port in2 and1))
            (subconn* (port cin full-adder) (port in2 xor2)
                                            (port in1 and2))
            (subconn  (port out xor2) (port sum full-adder))
            (subconn  (port out or1) (port cout full-adder))))
```

Higher level behavioral description for full-adders .

```
(protobehavior full-adder
 ((if (and (true (value (port in1 full-adder) $a) $t)
           (true (value (port in2 full-adder) $b) $t)
           (true (value (port cin full-adder) $c) $t))
      (and (true (value (port sum full-adder) (excl-or $a $b $c)) $t)
           (true (value (port cout full-adder) (carry $a $b $c)) $t)))))
```

Prototype Definition for adders.

```
(prototype (adder n)
  ((if (range $i 1 n) (subpart adder (num fa $i)))
   (if (range $i 1 n) (type (num fa $i) full-adder))
   (if (and (range $i 1 (1- n))
            (= $j (1+ $i)))
       (conn (port cout (num fa $i)) (port cin (num fa $j))))
   (if (and (range $i 1 n) (= $j (1- $i)))
       (if (and (true (value (port in1 adder) $v) $t)
                (= $b (bit $j $v)))
           (true (value (port in1 (num fa $i)) $b) $t)))
   (if (and (range $i 1 n) (= $j (1- $i)))
       (if (and (true (value (port in2 adder) $v) $t)
                (= $b (bit $j $v)))
           (true (value (port in2 (num fa $i)) $b) $t)))
   (if (and (true (value (port sum (num fa 1)) $a) $t)
            (true (value (port sum (num fa 2)) $b) $t)
            (true (value (port sum (num fa 3)) $c) $t)
            (true (value (port sum (num fa 4)) $d) $t)
            (= $e (integer $d $c $b $a)))
       (true (value (port out adder) $e) $t))))
```

Higher level behavioral description for adders.

```
(protobehavior (adder n)
 ((if (and (true (value (port in1 adder) $a) $t)
           (true (value (port in2 adder) $b) $t)
           (= $c (+ $a $b)))
      (true (value (port out adder) $c) $t))))
```

Prototype definitions for d74s.

```
(prototype (d74 n)
   ((subpart* d74 m1 m2 m3 a1 a2)
    (type m1 multiplier) (type m2 multiplier) (type m3 multiplier)
    (type a1 (adder n)) (type a2 (adder n))
    (conn  (port out m1) (port in1 a1))
    (conn* (port out m2) (port in2 a1) (port in1 a2))
    (conn  (port out m3) (port in2 a2))
    (subconn* (port in1 d74) (port in2 m1) (port in1 m2))
    (subconn* (port in2 d74) (port in1 m1) (port in2 m3))
    (subconn* (port in3 d74) (port in2 m2) (port in1 m3))
    (subconn  (port out a1) (port out1 d74))
    (subconn  (port out a2) (port out2 d74))))
```

Higher level behavior description for d74s.

```
(protobehavior (d74 n)
  ((if (and (true (value (port in1 d74) $a) $t)
            (true (value (port in2 d74) $b) $t)
            (true (value (port in3 d74) $c) $t)
            (= $d (+ (* $a $b) (* $a $c)))
            (= $e (+ (* $a $c) (* $b $c))))
       (and (true (value (port out1 d74) $d) $t)
            (true (value (port out2 d74) $e) $t)))))
```

Creating a single d74 called "foo" with 4 bit data paths.

```
(type foo (d74 4))
```

B. Transforming Prototype Definitions

This appendix will describe the transformations performed on the prototype definitions in order to uniquely name the subcomponents and simulate an instance at the appropriate level. For example, the protype definition for the parameterized adder is converted to the following internal form:

```
(IF (AND (TYPE |$$| (ADDER $N)) (RANGE $I 1 $N))
    (SUBPART |$$| (PART (NUM FA $I) |$$|)))

(IF (AND (TYPE |$$| (ADDER $N)) (RANGE $I 1 $N))
    (TYPE (PART (NUM FA $I) |$$|) FULL-ADDER))

(IF (AND (TYPE |$$| (ADDER $N))
         (RANGE $I 1 (1- $N))
         (= $J (1+ $I)))
    (CONN (PORT COUT (PART (NUM FA $I) |$$|))
          (PORT CIN (PART (NUM FA $J) |$$|))))

(IF (AND (TYPE |$$| (ADDER $N))
         (SIMSUB |$$|)
         (RANGE $I 1 $N)
         (= $J (1- $I)))
    (IF (AND (TRUE (VALUE (PORT IN1 |$$|) $V) $T)
             (= $B (BIT $J $V)))
        (TRUE (VALUE (PORT IN1 (PART (NUM FA $I) |$$|)) $B) $T)))

(IF (AND (TYPE |$$| (ADDER $N))
         (SIMSUB |$$|)
         (RANGE $I 1 $N)
         (= $J (1- $I)))
    (IF (AND (TRUE (VALUE (PORT IN2 |$$|) $V) $T)
             (= $B (BIT $J $V)))
        (TRUE (VALUE (PORT IN2 (PART (NUM FA $I) |$$|)) $B) $T)))

(IF (AND (TYPE |$$| (ADDER $N))
         (SIMSUB |$$|)
         (TRUE (VALUE (PORT SUM (PART (NUM FA 1) |$$|)) $A) $T)
         (TRUE (VALUE (PORT SUM (PART (NUM FA 2) |$$|)) $B) $T)
         (TRUE (VALUE (PORT SUM (PART (NUM FA 3) |$$|)) $C) $T)
         (TRUE (VALUE (PORT SUM (PART (NUM FA 4) |$$|)) $D) $T)
         (= $E (INTEGER $D $C $B $A)))
    (TRUE (VALUE (PORT OUT |$$|) $E) $T))
```

```
(IF (AND (TYPE |$$| (ADDER $N))
         (SIMTOP |$$|)
         (TRUE (VALUE (PORT IN1 |$$|) $A) $T)
         (TRUE (VALUE (PORT IN2 |$$|) $B) $T)
         (= $C (+ $A $B)))
    (TRUE (VALUE (PORT OUT |$$|) $C) $T))
```

REFERENCES

1. Aho, Alfred E, et. al., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Menlo Park, 1976.
2. Barnes, T., *"BAL: A Bozes and Lines Editor for Design,"* Heuristic Programming Project Memo, Stanford University, 1983. In preparation.
3. Brown, H., et. al., *"Palladio: An Exploratory Environment for Circuit Design,"* Heuristic Programming Project Memo HPP-83-31, Stanford University, 1983.
4. Foyster, G., *"Behavioral Specification and Verification within a VLSI Design System,"* Heuristic Programming Project Memo HPP-83-16, Stanford University, 1983.
5. Genesereth, M., *"The MRS Manual,"* Heuristic Programming Project Memo HPP-80-24, Stanford University, 1980.
6. Genesereth, M., *"SUNDEW: The Stanford University Design Entry Workstation,"* Heuristic Programming Project Memo, Stanford University, 1983. In preparation.
7. Nilsson, N. J., *Principles of Artificial Intelligence,* Tioga Press, Palo Alto, 1980.
8. Pitman, K. M., *The Revised Maclisp Manual,* Laboratory for Computer Science Memo MIT/LCS/TR-295, Massachusetts Institute of Technology, 1983.