USES OF TIME INTERVALS TO MODEL DIGITAL BEHAVIOR

by

Michael Roger Crystal

Submitted to the Department of
Electrical Engineering & Computer Science
in Partial Fulfillment of the Requirements
for the Degrees of

Bachelor of Science
&
Master of Science

at the

Massachusetts Institute of Technology
June 1987

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science
May 08, 1987

Certified by Signature redacted
Dr. Charles Rich
Thesis Supervisor, Principal Research Scientist

Certified by Signature redacted
Glenn A. Kramer
Company Supervisor

Accepted by Signature redacted
Prof. Arthur Smith
Chairman, Department Committee on Graduate Students

Uses of Time Intervals to Model Digital Behavior

Michael R. Crystal

May 6, 1987

USES OF TIME INTERVALS TO MODEL DIGITAL BEHAVIOR

by

Michael Roger Crystal

Submitted to the Department of Electrical Engineering & Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science & Master of Science

ABSTRACT

I present a series of case studies that demonstrate that an interval representation of time is more expressive than a point representation for digital design reasoning tasks. The case studies exemplify where the use of time intervals results in increased modeling efficiency, where it results in increased modeling accuracy, and where it enables new tasks to be performed. To provide a basis for comparison, I have modified an existing digital design description language to include a vocabulary for referencing time intervals.

Thesis Supervisor:	Dr. Charles Rich
Title:	Principal Research Scientist, MIT
Thesis Supervisor:	Glenn A. Kramer
Title:	Member Technical Staff, Schlumberger Palo Alto Research

ACKNOWLEDGMENTS

Many people have helped me while I was working on this thesis. At Schlumberger, my supervisor Glenn and colleagues Narinder, Jeff, Don, Saied, and Kim, through many discussions have added their insights, and in general made the lab an enjoyable place to work. At MIT, my advisor Chuck not only read my thesis, but also taught me about the field of AI research.

I am grateful to my friends Erik and Carol. Carol read over one of the first drafts of the thesis, and the last; the role her encouragement played in the final product cannot be overestimated.

Lastly, I would like to thank my family for their continual love and support.

Contents

1 Introduction **7**

1.1 Previous Work 8

1.2 The Thesis 9

1.3 Overview 10

2 Context — The Helios Design System **11**

2.1 Corona 12

2.2 MARS 15

3 Interval Reasoning System **21**

3.1 The `Ival` Predicate 21

3.2 Other Predicates 25

3.2.1 IDelay 25

3.2.2 ILength 26

3.2.3 Meets 27

3.2.4 Intersect 29

3.3 MARS Modifications 30

3.4 An Example 33

3.4.1 MARS & The Point Model 35

3.4.2 IMARS & The Interval Model 36

3.4.3 Comparison & Contrast 39

4 Examples **41**

4.1 Correctness 42

4.1.1 Setup Time 42

4.1.2 Consecutiveness 46

4.1.3 Tristate Devices 51

4.2 Efficiency 52

4.2.1 Latch 53

4.2.2 Recheck 58

4.3 New Tasks 60

4.3.1 Contention 60

4.3.2 No Behavior 62

4.4 Subsumption 63

5 Summary & Conclusion **65**

List of Figures

2.1 Examples of Corona Statements 12

2.2 A partial description of an OR gate with delay. 13

2.3 The MARS Algorithm 17

2.4 A D flip-flop 18

2.5 Simulation of a D Flip-Flop 19

3.1 A Nand Gate 34

4.1 Point Model of a Latch with a 20ns Setup Time 43

4.2 Interval Model of a Latch with a 20ns Setup Time . . . 47

4.3 Point Model of an Edge-Triggered D Flip-Flop 48

4.4 Interval Model of an Edge-Triggered D Flip-Flop . . . 50

4.5 Point Model of a Two State Machine 54

4.6 Interval Model of a Two State Machine 56

4.7 Point Model of Device Requiring a Recheck Marker . . 58

4.8 **Simulation of Zero Delay Components** 61

Chapter 1

Introduction

The choice of a good representation language in which to do problem solving is a central issue in design [Ama81]. Kramer demonstrates that changes in vocabulary can result in exponential reductions in the computational complexity of some reasoning tasks [Kra87].

In this thesis, I compare and contrast two models of time for digital design simulation: (a) a point-based or discrete model, and (b) an interval-based or continuous model. The results of the thesis are a series of case studies that demonstrate techniques for and advantages of using the interval representation.

A point model is currently used in the Helios Design Assistant [Kra85], wherein an event is defined as the change in the value of some object. The occurrence of the event is specified by stating the new value of the object, and the time at which the change occurs. For example, the output of a counter becoming 3 at time 5 is an event. The behavior of an object is represented by a series of pairs of events and the points in time at which they occur.

In the interval model, an event is any *unchanging* value that an object has: An event is defined as a value and the interval of time over which that value does not change. An object's behavior, using this model, is represented as a series of pairs of events, and the intervals of time over which they exist.

Filling a tub might be represented by: (a) The tub is empty from time 0 to time 3; (b) From time 3 to time 5 the tub is filling at a constant rate of 2 cubic feet per second; and (c) from time 5 to time ∞ the tub contains 4 cubic feet of water.

To view the point model more formally, an isomorphism can be constructed between the time point domain and the set of real numbers. In cases where time is assumed to have an absolute starting point, as opposed to extending back infinitely, the isomorphism is limited to the non-negative reals. A time point corresponds to a point on the number line.

Extending this analogy, intervals are convex sets of time points. Less formally, an interval is any piece of the real number line that does not contain a hole.

From this analogy one can infer that there is a qualitative difference between a time point, and an interval. All non-trivial (*i.e.* those which are neither empty nor singleton) intervals are made up of an infinite number of points. Unquantified first order predicate calculus statements about intervals cannot be made using a finite number of statements from the point domain.

1.1 Previous Work

The idea of using intervals of time, instead of points, to describe when events occur is neither new nor limited to the digital design domain. Allen presents 13 disjoint, yet complete relations[1] that can exist between any two intervals of time [All83]. These make up his "interval-based temporal logic".

Along with the logic, he presents an $O(n^2)$ time, $O(n^2)$ space algorithm for deducing what relations can exist between two intervals. "It should be noted that this algorithm, while it does not generate inconsistencies, does not detect all inconsistencies in its input. ...the computational complexity

[1]The relations are *complete* because there is at least one relation which holds between any two intervals, and are *disjoint* because no more than one relation holds between any two intervals.

of the algorithm [to detect all inconsistencies] is exponential." [All83, p.837]

Many papers extend Allen's work. Hayes and Allen use the first order predicate calculus to axiomatize the 13 relations in terms of a single relation, *Meet* [AH84]. Recently, Ladkin has supplemented Allen's work by developing "a taxonomy of important binary relations between intervals which are unions of convex intervals" [Lad86a]. Ladkin describes the relations and situations in which they may be used [Lad86a,Lad86b].

Within the domain of digital design, Moszkowski developed a Propositional Interval Temporal Logic (ITL) [Mos83]. He formalizes ITL, presents a complexity analysis, and then shows how ITL may be used to model a variety of simple digital devices.

Joyce takes a single example of using intervals to model digital designs and contrasts "two techniques for representing time-dependent digital system behavior and controlling reasoning to achieve desired hardware states," [Joy83]. He concludes that by using intervals and propagating constraints, diagnostic reasoning in the digital design domain can be done more efficiently.

1.2 The Thesis

A new representation has been developed for modeling the time at which events occur. I present a series of case studies that demonstrate the increase in representational power one attains by switching from the point model to the interval model of time.

I have divided the quality of representational power into three components: modeling accuracy, modeling efficiency, and performable tasks. Modeling accuracy is concerned with the extent to which the design vocabulary allows the user to describe the actual behavior of the device being designed.

Modeling efficiency relates to the tasks which one would want to perform with a model. Certain vocabularies or representations lend themselves to efficient performance of certain tasks.

A natural extension to task efficiency is task performability. The representational power of one language is greater than that of another if there exist certain tasks which can be performed in the former, but not the latter.

I have implemented an Interval Reasoning System (IRS) that extends Corona [Sin84], an existing design language, so that an event may be described as taking place during an interval of time. IRS differs from Allen's work in several areas. IRS is not a general interval problem solver, but rather a vocabulary tailored to aid in digital design related tasks. The relations available for comparing intervals are not complete; however, none of the associated algorithms work in worse than $O(n)$ time, or $O(n)$ space. The work presented in this thesis has been implemented using IRS.

1.3 Overview

The remainder of this thesis is organized as follows: Chapter 2 presents the context in which the work was done, The Helios Design Assistant. Chapter 3 begins with definitions of an interval and the associated IRS predicates. It then includes a discussion of the changes that were made to the Helios simulator to accommodate the interval vocabulary. Chapter 3 ends with a comparison and contrast of two behavioral descriptions of a simple circuit element. The first description uses the point model of time, the second uses the interval model. Chapter 4 presents case studies which demonstrate how intervals increase model accuracy and design task efficiency, and how new tasks may be performed as a result of the introduction of an interval vocabulary. Behaviors that can be modeled more accurately with intervals include setup time, consecutive events, and tristate devices. Increased reasoning efficiency is demonstrated on a latch, and a generic device with propagation delay. New tasks that can be performed with intervals include detecting contention and detecting when an object does not have a defined behavior. Chapter 5 summarizes the results of the case studies and presents directions for future work.

Chapter 2

Context — The Helios Design System

IRS operates within the context of the Helios Design Assistant [Kra85]. Helios's goal is to provide an environment in which a declarative behavioral description of an artifact may be entered once, and several tasks may then be performed using the single design description. Currently the Helios subsystems include a simulator, a test vector generator, a verifier, a diagnostician, an input editor, and an embedded auto-router.

Single design entry offers many advantages. The most apparent is the time savings for the design engineer. Another is consistency. One of the most difficult types of design errors to locate is that arising from the use of inconsistent behavioral models for different tasks during the design process. One requirement of IRS is that it must support the single model principle.

To allow for the explicit reasoning which a design system must perform, Helios uses MRS [Rus85]. MRS is logic programming substrate that uses pattern matching to store, access, and make deductions with the behavioral descriptions of artifacts under design.

From a control viewpoint, there are two categories of MRS statements. The first consists of unquantified first order predicate logic statements. Sen-

Rule Type	*Corona*	*English*
Structure	`(port A B)`	A is a port of module B.
Behavior	`(val A 1)`	A has the value 1.
	`(true (val A 1) 10)`	The statement, "A has the value 1" is true at time 10.
Predicate	`(bnot A B)`	The boolean negation of A is B.

Figure 2.1: **Examples of Corona Statements**

tences are clauses with universal quantifiers removed and existential quantifiers handled through Skölem functions.

The second type are meta-level control statements. These are rules which allow for explicit reasoning about the base level sentences. Meta-level statements can modify how lookups and assertions are done, and control the order in which they are done. Several of the Helios subsystems are written in MRS using the meta-level statements.

2.1 Corona

Helios behavioral descriptions are written in the language Corona [Sin84]. Corona is a set of predefined MRS predicates that are useful for specifying behaviors of digital designs. Corona does not mask any of the MRS constructs or capabilities.

The three major types of Corona statements are listed in figure 2.1. The first is for describing the structure of artifacts. The second describes the behavior of artifacts. The third type of statements includes generally useful predicates such as boolean not. So that all possible instantiations do not need to be stored, some of the predicates are implemented with procedural attachments.

In general, a Corona behavioral description is given as a set of "if-then" rules which include all three types of statements. The if-then rules are specified by the Corona `if` statement. The `if` statement takes two ar-

```
(if (and (true (val (port in1 or-gate) on) $t)
         (+ $t 5 $t1))
    (true (val (port out or-gate) on) $t1))
```

Figure 2.2: **A partial description of an OR gate with delay.**

guments, an antecedent and a consequent. The antecedent is either a single Corona statement, or several statements conjoined by the `and` connective. The consequent has the same syntax. A "$" before a variable name indicates a universally quantified variable.

The example in figure 2.2 partially describes an OR gate with a propagation delay of 5 time units: If at time `$t` the value of the `in1` port of device `or-gate` is `on`, and if there is a value for `$t1` such that the statement `(+ $t 5 $t1)` is true, then the value of the `out` port of `or-gate` is `on` at time `$t1`.

The Helios requirement that all tasks can be performed using a single description restricts what predicates may exist in Corona. Notice that all of the predicates in figure 2.2 represent invertible functions. It is just as easy to compute `$t1` given `$t` as it is to go in the other direction. This need arises because certain systems naturally use backward reasoning —given the consequent, determine what antecedents must be present. An example of a backward reasoning system is the Helios diagnostician, DART [Gen81]. The inputs to DART are a set of supposed faulty outputs to a real device. DART works backwards from the faulty outputs to determine what rules in the model, if not adhered to, account for the behavior.

The depth first inference mechanism that is most often used for reasoning with Corona rules also restricts the set of viable Corona predicates. To derive a consequent, this mechanism steps through the conjuncts of the antecedent. When it reaches an uninstantiated variable, it finds a possible instantiation of the variable that makes the predicate in which it occurs true. If a conjunct is not satisfiable, then the mechanism backtracks to the last point a variable occurred, and selects a different possible instantiation.

Because the MRS inference mechanism binds variables as they occur, the domain of values for unbound variables must be restricted to a small set by the bound variables in the enclosing predicate. Otherwise, the probability of finding the binding that satisfies other conjuncts in the same rule is low, resulting in poor efficiency. Predicates in which knowledge of all variable bindings except one do not restrict the possible values for the unbound variable are called `non-invertible`.

An example of a non-invertible predicate is $>$. It could be reasonable in some situation to replace `(+ $t 5 $t1)` by `(> $t1 $t)`. Knowing `$t` would offer little in determining `$t1`. If there were a small set of `$t1`'s that satisfy latter constraints, then the system could go into an infinite loop choosing a bad instantiation and backtracking.

Because there are times that relations such as $>$ are necessary, they are included in Corona. To avoid the problems described above, a non-invertible predicate may not have any uninstantiated variables when it is looked up. Joyce's Achiever [Joy83] investigates the effects of delaying variable instantiation. Instead of choosing a single possible instantiation of a variable, instantiating the variable, and moving to the next conjunct in the antecedent, Achiever is able to instantiate underspecified variables to sets of inequalities. "The bulk of the current instantiator is a fairly simple algorithmic inequality solver based on work by Markowsky." [Joy83, p.22].

MRS, and subsequent systems built on top of it also suffer from the problems that classically plague logic-based languages. For example, it is extremely inefficient to reason using syllogism. Encoding the knowledge that, "Exactly one of the outputs of a 2-to-4 decoder is valid at any one time," is very difficult unless it is done by enumerating the possible output combinations. To capture the meaning of the above statement one must be able to say, "If it is *not* the case that one of the first three outputs is high, then the fourth is high." But, checking whether a statement is *not* the case requires that the system tries to prove the statement in all possible ways, and is unable to. Corona does have a predicate, `unknown`, which does this, however its use is extremely inefficient, and its use is discouraged.

Another important point to make is that although the `true` predicate is treated specially by the systems that use Corona, it is just an ordinary binary predicate. `(True <a><b>)` may be asserted, and looked-up in the same way that `(Father Abraham Isaac)` is. `True`'s arguments may be anything. However, to retain the semantics of the predicate, the first argument is expected to be another statement, and the second argument is a real number. A consequence of the `true` statement having no special meaning is that something asserted to be `true` at time 10 has no bearing on time 10.5. This is an instance of the more general problem of determining what facts should persist when the world changes state. Hayes called this the *frame problem* [Hay73].

The frame problem is especially relevant when time is modeled as a continuous flow. If time is considered discrete, values that persist can be reasserted at the next step in time. The axioms which do the reassertion are called *frame axioms*. In the case where time is continuous, there is no *next* step at which a fact can be reasserted.

Corona, however, ignores the frame problem. The choice of what frame axiom to use is left to the individual Helios subsystems, and is not necessarily the same for each. In all cases, the choice represents an implicit assumption that must be made by the subsystem. Using only a point vocabulary there is no way for the user to explicitly describe a frame over which a behavior persists.

2.2 MARS

The system on which all of the case studies have been run is MARS [Sin83], the Helios simulator. "The basic inference technique used by MARS for performing event-driven simulation is *forward chaining* on an explicit rule base." [Sin83, p.14] Although not completely accurate, MARS can be thought of as a collection of MRS meta-level rules, which affect the way assertions and lookups of the `true` predicate are made.

One of the effects of the MARS meta-level rules is that when `true` statements are first asserted they are placed in a heap indexed by their second argument. Recall that the second argument to the `true` statement is a real number representing a point in time. MARS expects the first argument to be a statement of the form, `(val (port <a> <b>) <c>)`. The `true` statement with the minimum second argument value is at the top of the heap. This value is held in a global variable called `current-time`. Just as events in the future can not cause events in the past, no `true` statement may be asserted whose second argument is less than `current-time`.

After a `true` statement has been asserted it is stored in a tree that is also indexed by the second argument of the `true` statement. There is a tree associated with each port of each device in the design. The tree contains the chronological history of the port with which it is associated. The purpose of storing `true` statements in this manner is that lookups may be performed in $O(\ln n)$ time in the number of `true` assertions which have been made about that port.

The skeleton MARS algorithm is given in figure 2.3. MARS loops as long as the scheduling heap is not empty. For each loop, MARS removes the first statement in the heap and asserts it. As a result of the assertion, other statements may be placed in the heap.

Figure 2.4 depicts a rule which models a D flip-flop with a 5 time-unit delay: If the `clk` port of the D flip-flop `d-ff` is 1 at time `$t1` and was 0 one time unit previous to `$t1`, then 5 time units after `$t` the `q` port of `d-ff` will have the value that the `d` port has at time `$t`.

As an example, assume that in addition to the D flip-flop rules from figure 2.4, the database contains the assertions:

```
(true (val (port clk d-ff) 0) 0)
(true (val (port d d-ff) 1) 1)
```

And the heap contains the single entry:

```
(true (val (port clk d-ff) 1) 1)
```

1. Starting at the top of the heap, if the first argument to the `true` statement has been previously asserted and no different statement has been made about the same port since the original statement, remove the `true` statement and repeat step 1.

2. Remove the top `true` statement from the heap, and make a list of all rules from the database that contain a conjunct in their antecedent that matches the removed statement.

3. For each of those rules, taken in arbitrary order:

   (a) Unify the removed statement with the conjunct that it matches, and create an environment which has all free variables that were matched in the unification bound to the value that they matched.

   (b) For each conjunct, in the order they are listed, from the antecedent of the rule:

      (i) Instantiate the values of the variables which have bindings from the enclosing environment.

      (ii) Do a lookup on the resulting expression.

      (iii) If the lookup returns no matches from the database, then the rule fails; go on to the next rule.

      (iv) If the lookup returns exactly one match then update the enclosing environment to include the new bindings, and proceed to the next conjunct.

      (v) If the lookup returns `n` matches, where $n > 1$, then create `n` environments in the same manner as the previous step, and continue this algorithm on each of the new environments independently.

   (c) Using the new environment, instantiate the unbound variables in each of the conjuncts of the consequence of the if-statement.

   (d) Assert each of the instantiated conjuncts in the consequence by placing them in the heap.

4. Add the `true` statement to the history of the port mentioned in the second argument of the `true` statement.

Figure 2.3: **The MARS Algorithm**

```
(if (and (true (val (port clk d-ff) 1) $t1)
         (+ $t 1 $t1)
         (true (val (port clk d-ff) 0) $t)
         (true (val (port d d-ff) $d) $t)
         (+ $t 5 $t2))
    (true (val (port q d-ff) $d) $t2))
```

Figure 2.4: **A D flip-flop**

Figure 2.5 presents a step by step record of the simulation. There are several details of the algorithm brought out by this example that deserve special mention. First, MARS unifies the conjuncts of the antecedent in the order that they are specified in the rule. If the assertion that causes the rule to fire is the clock becoming 0, `(true (val (port clk d-ff) 0) 4)`, then when the algorithm looks up the first conjunct, it returns all times at which `clk` is 1. If the D flip-flop were part of a real time device that contained a micro-second clock, the number of possible matches returned could easily overburden MARS.

To cope with this situation a special predicate, `+*`, was added to Corona which could be placed as the first antecedent of a rule, and does all of the necessary arithmetic before any other lookups are done. This scheme avoids the problem of a time variable being unbound by using the time variable from the conjunct that caused the rule to fire, to calculate the other time variables. For the case above, `+*`, given any of `$t, $t1,` or `$t2`, computes the other two.

Another key point to note in the operation of MARS is how it deals with simultaneous events. Singh [Sin85, p.63] presents a simple situation

1. The single entry in the heap is removed from the top. This `true` statement is matched against the first conjunct in the only rule (*i.e.* if-statement) in the database.

2. An environment is created with `$t1` bound to 1.

3. The first conjunct is re-looked up.

4. In the second conjunct of the antecedent, `$t1` is replaced by 1, and the result is looked up. The result of the lookup causes the binding, `$t` to 0, to be appended to the environment. Notice that the predicate can solve the addition both backwards and forwards.

5. The `$t` in the third conjunct is replaced by 0, and the fact, `(true (val (port clk d-ff) 0) 0)` is looked up. It is found in the database so the algorithm proceeds.

6. The rest of the conjuncts in the antecedent cause `$t2` to be bound to 6 and `$d` to be bound to 1.

7. Instantiating the free variables in the consequence of the rule with the bindings generated from the antecedent yields `(true (val (port q d-ff) 1) 6)`. This fact is asserted and inserted into the heap.

8. The scheduler removes the top assertion from the heap, the statement just asserted, and finds that it matches no rules. The heap is empty so the simulation is finished.

Figure 2.5: **Simulation of a D Flip-Flop**

in which MARS, to simulate simultaneous events correctly must overwrite one assertion in a port's history with another. In short, if there exists an assertion of the form, `(true (val (port x y) z) t)`, and a new assertion is made equivalent to the first except the value `z` is different, then the original assertion is removed from the port `x`'s history and replaced with the second.

Because the vocabulary MARS uses to describe time is limited to points, MARS is limited in the way in which it can deal with the frame problem. There is no way to describe a frame over which something is explicitly stated to be true. Consequently, MARS implicitly assumes the *persistence* of values once they have been asserted. So, for example, if `(true (val (port a b) c) t)` had been asserted, then a lookup done on `(true (val (port a b) $x) t1)` causes `$x` to be bound to `c`, as long as `t1` after `t`, and no other statement was made about the value of port `a` of device `b` since `t`.

Persistence is implemented in two places in the MARS algorithm. In step 1, if the previous value a port was asserted to have is the same as the value currently asserted, then the current assertion is removed from the heap with no side effects. In step 3b, the algorithm for doing a lookup is modified slightly to implement persistence. If a port does not have a value that explicitly was given to it at a certain time, (that is there is no perfect match in the database) then the lookup routine searches backwards in the tree to find the previous time at which a value was asserted to exist at the port. This value is assumed to persist to the time currently being looked up.

Because assertions may not be made about events in the past, and lookups may not be made of events in the future, there is no possibility that a value will be asserted between the current simulator time and the time that something was previously asserted.

Implicit persistence avoids the problem of requiring a method for deciding what ports will retain their values to the current time. As some of the case studies in chapter 4 show, it also limits what the user can say. Much of the power of a vocabulary that can explicitly reference intervals is derived from the ability of an interval to represent a frame during which a value on a port persists.

Chapter 3

Interval Reasoning System

In the point model, the behavior of a device is represented by a sequence composed of pairs of events and the times at which they occur. The `true` statement in MARS creates these pairs out of its first and second arguments. In the interval time model, the behavior of a device is represented as pairs of constant behaviors and the intervals over which the behaviors occur. To accomplish the pairing I have written IRS. IRS is an extension to Corona that includes useful predicates for manipulating intervals.

3.1 The `Ival` Predicate

The syntax of `ival` is analogous to that of `true`. `Ival` takes two arguments. The first is a statement, and the second, instead of a point, is an interval. `(Ival (val (port d d-ff) 1) <interval>)` asserts that the `d` port of device `d-ff` has the value 1 during the interval `<interval>`.

An interval is a 4-tuple represented by a list of 4 elements. The first value is either closed (c) or open (o). This indicates whether the lower bound of the interval is included or excluded from the interval. The second element of the 4-tuple is any real number from $-\infty$ to ∞. The third and fourth elements of an interval are a mirror image of the first two. The third specifies the upper bound of the interval, and the fourth specifies whether it is open or closed. In addition to any real number, the third element of the tuple also can be the symbol `persists`.

The interval syntax is constrained by two global requirements:

1. The lower bound of the interval must be less than or equal to the upper bound.
2. If the lower and upper bounds of the interval are equal, then both the upper and lower bounds of the interval must be closed.

Examples of intervals are:

```
(o 2 5 o)
(c 4 4 c)
(c 2 persists o)
```

When the upper bound of an interval is not `persists`, the semantics of the 4-tuple is straightforward and best demonstrated by example. The first example from above is the interval from 2 to 5 exclusive. The second is the interval from 4 to 4, or more simply, the point 4. The second example demonstrates how the interval logic degrades nicely into the point representation.

A more formal definition of an interval is:

<c i j o> $\equiv \{x : i \leq x < j\}$,
and similarly for the other three cases.

This expands easily to yield a natural semantics for the `ival` statement:

`(ival (val (port d d-ff) 1) (c i j o))` $\equiv$
{(true (val (port d d-ff) 1) x): $i \leq x < j$}

As with `true`, the first argument to the `ival` statement is any other legal Corona statement. This means that it can contain constants or variables. Because the second argument of the `ival` statement is used for indexing, when `ival` is **asserted**, none of the elements of the interval may be variables.

Using the initial definition of `ival` , lookup is not well-defined. This is because of the infinity of points represented by an interval. For example, if after `(ival (val (port d d-ff) 1) (c 2 5 o))` is asserted a lookup is done on `(ival (val (port d d-ff) $x) (o 3 4 o))`, to what should $x be bound? There is no fact in the data base that matches what is being looked up.

This problem is solved by introducing the predicate `contains`. `(Contains` <a> <b>`)` is true if and only if the interval <b> is completely contained within the interval <a>. `Contains` is formally defined as follows:

$$(\texttt{contains}\ \langle a \rangle\ \langle b \rangle) \equiv \forall x.[(x \in \langle b \rangle) \rightarrow (x \in \langle a \rangle)]$$

Now, for the purposes of a lookup, an interval can unify with another interval that contains it. One of the consequences of this is that lookups are no longer really doing unification, what is being done is not commutative. Interval `a` can unify with interval `b` even though `b` does not unify with `a`.

Another ambiguity arises when an unknown occurs in the second argument of the `ival` statement being looked up. Again, let `(ival (val (port d d-ff) 2) (c 2 5 o))` be a fact in the data base. This time the expression, `(ival (val (port d d-ff) 2) (c 2 $x o))` is to be looked up. The obvious choice for $x is 5, however, applying the aforementioned `contains` predicate, 4 is an equally valid, and possibly more desirable choice for a binding.

To make an `ival` lookup well-defined, I have adopted the principle of always binding the largest possible subinterval to an unknown. There are several justifications for this. First, it resolves the previously described ambiguity associated with lookups. There is almost always[1] a unique largest subinterval that is contained by a given interval, and satisfies a given set of constraints.

Second, only the largest subinterval contains enough information to generate all other subintervals that match the constraints. This follows from the definition of an interval as a convex set of points.

Kramer also proposes the strategy of starting to reason with the largest intervals as a means of reducing the number of entities considered during problem solving [Kra84, p.92]. He claims that this is an instance of hierarchical decomposition used for problem solving.

The `persists` Marker

When one considers a single interval, the `persists` marker has little meaning. Applying the rules that define a well-formed interval, it can be inferred that the `persists` marker represents a real number that is greater than or equal to the lower bound of the interval in which it occurs.

As part of a chronological series of intervals, such as the one used to store the history of a port, the `persists` marker is used to denote an upper bound that is as large as possible without conflicting with the assertion over the next interval of time, *i.e.* the point time semantics of `true`. Modeling memory is an instance in which one would want to use the `persists` marker. The value of a memory slot persists until it is changed.

To use the `persists` marker some concept of a *next* interval must exist. This requires that intervals be well-ordered. Allen introduced 13 relations that can exist between any two given intervals [All83]. In general, a total ordering cannot exist if more than two relations, $>$ and its inverse, $<$,

[1]Cases still exist wherein an ambiguity can arise. For example, if the interval in the database is `(o 2 5 o)` and the interval being looked up is `(o 2 $x c)`, there does not exist a largest interval that matches the database entry and satisfies the constraints placed by the interval being looked up. Because of the continuity of real numbers, open ended sets do not contain a largest element. In this case an `ival` lookup fails.

exist between any two unique objects in the domain.

Because the lower bound of an interval is always a real number a partial ordering of intervals may be formed using the lower bound. If two intervals have the same lower bound, the one whose first element is `c` is less than the one whose first element is `o`. This definition corresponds to ordering intervals by their starting points.

None of the intervals that describe the behavior of a device overlap. If two intervals overlap and the events they are paired with are the same, then there is a single larger interval that corresponds to the time over which the constant behavior occurs. If the events are different, then there is an inconsistency: At some point in time the device has two different behaviors.

Because the intervals that describe the behavior of a specific device do not overlap the partial ordering between intervals described above is adequate for generating a unique ordering of the intervals in the history of a device. Consequently, when one is talking about the behavior of a device there exists the concept of a *next* interval, and the `persists` marker has a well-defined meaning.

3.2 Other Predicates

Along with the `ival` statement for asserting and looking up statements which hold over intervals, IRS provides four predicates for manipulating intervals. They are: `IDelay`, `ILength`, `Intersect`, and `Meets`.

3.2.1 IDelay

$$\texttt{(IDelay}\ \mathrm{n}\ <\!ival_1\!>\ <\!ival_2\!>) \equiv \forall \mathrm{x}.[(\mathrm{x} \in <\!ival_1\!>) \Leftrightarrow (x + n \in <\!ival_2\!>)]$$

`IDelay` is the interval delay operator. It takes 3 arguments, a real number, and two intervals. Just as 2 of the 3 arguments to + must be specified to do a lookup, enough information must be given to `IDelay` so that it may uniquely determine all unbound variables. If there is no unique solution, `IDelay` is `false`.

`IDelay`'s principal use is to model the propagation delay associated with a device. In figure 2.2 a partial description of an Or gate with a propagation delay of 5 time units was given. Recast in the interval domain, the behavior is modeled by:

```
(if (and (ival (val (port in1 or-gate) on) $t)
         (idelay $t 5 $t1))
    (ival (val (port out or-gate) on) $t1))
```

The only differences between the two models are that `true` is replace by `ival`, and + is replaced by `IDelay`.

3.2.2 ILength

(`ILength` n <a_0,a_1,a_2,a_3>) $\equiv$ $a_1 + n = a_2$

`ILength` pairs an interval and a real number that represents the length of the interval. Notice that there are no restrictions on the values of a_0 and a_1. Eventhough, there is no constraint on their values, the values must be specified for a lookup or assertion to be true. There is no analogous predicate in the point domain because there is no concept of a point having a width. As with `IDelay`, enough information must be given to `ILength` for it to bind all unbound variables, otherwise it is `false`.

The most common use of `ILength` is to insure that minimum requirements, such as setup time, are adhered to. The following is the same Or gate as in the previous example, except an additional requirement is added: The signal on port `in1` of device `or-gate` must be steady for at least 5 time units if a change is to occur on the output.

```
(if (and (ival (val (port in1 or-gate) on) $t)
         (idelay $t 5 $t1)
         (ilength $n $t)
         (> 5 $n))
    (ival (val (port out or-gate) on) $t1))
```

The > relation and the `ILength` predicate could have been combined to form a single minimum interval length relation, however, the new relation would suffer from the same problem as <: All of the arguments to the new relation would have to be bound before a lookup could be done.

When modeling hold-time (the interval of time over which an output remains steady after it has been asserted), one wants to go from a partially specified interval and an interval length to the completely specified interval. If the minimum length relation existed instead of `ILength`, then such a computation would not be possible.

3.2.3 Meets

```
(Meets n <$a_0$,$a_1$,$a_2$,$a_3$> <$b_0$,$b_1$,$b_2$,$b_3$>) ≡
      (∨ (∧ (= n $a_2$ $b_1$)
            (∨ (∧ (= $a_3$ c) (= $b_0$ o))
               (∧ (= $a_3$ o) (= $b_0$ c))))
         (∧ (= n $b_2$ $a_1$)
            (∨ (∧ (= $b_3$ c) (= $a_0$ o))
               (∧ (= $b_3$ o) (= $a_0$ c)))))
```

`Meets` is one of the 13 relations that Allen specified could exist between two intervals [All83]. The first argument to `Meets` is a point. The second and third arguments are intervals. `Meets` is true if and only if the two intervals meet at the point. Two intervals meet if they do not overlap and there does not exist a point which is less than all of the points in one interval and greater than all of the points in the other. For example:

`(Meets $pt (o 2 5 c) (o 5 7 o))` is true, and binds \$pt to 5.
`(Meets $pt (o 2 5 c) (c 5 7 o))` is false
—there exists a point of overlap.
`(Meets 5 (o 2 5 c) (o $low 7 c))` is true,
and binds \$low to 5.

If there is not adequate information to bind all of the unbound variables then `Meets` is `false`. If this were not the case, `Meets` would have an infinite number of possible bindings. A rule using `Meets` could loop infinitely between choosing a binding for the variables in `Meets`, and failing in subsequent conjuncts of the same rule.

The following example is false because \$x may take on any value greater than 5:

`(Meets $pt (o 2 5 c) (o 5 $x o))` is false.

Meets with Ports

In practice, `Meets`, as it is described above, has limited usefulness. This is because of the requirement that `Meets` can bind all of its variable arguments. Often, only one of the interval arguments to `Meets` is known when `Meets` is applied. This problem is solved by noticing that the interval arguments are almost always elements of the histories of ports. By pairing a port name with each of the interval arguments, `Meets` has a finite domain, the history of the port, in which it can search for bindings for unspecified intervals. The syntax of `Meets` with ports is:

(Meets n (<*port-name*$_1$> <*interval*$_1$>) (<*port-name*$_2$> <*interval*$_2$>))

The use of `Meets` with ports is exemplified in the following rule:

```
(if (and (meets $t (in $t0) (in $t1)))
         (ival (val (port in change-detector) 1) $t1)
         (ival (val (port in change-detector) 0) $t0)
    (ival (val (port out change-detector) 1) (c $t $t c)))
```

If asserting that `in` has the value 1 causes the rule to fire, then when `Meets` is looked up, only one of the three variables is bound. For `Meets` to be true, `$t1` and `$t0` must share a boundary. Consequently, the problem of binding `$t0` is broken down into four cases. For each of these cases one boundary of `$t0` is known. With the boundary point fixed, there is at most one longest interval in the history of `in` that matches `$t0`. With ports, `Meets` is guaranteed to find a successful variable instantiation in a finite time, or fail because one does not exist.

3.2.4 Intersect

(`Intersect` <interval>$_0$ <interval>$_i^+$) $\equiv$

$$<interval>_o = \bigcap_i <interval>_i$$

The `Intersect` statement is true if and only if the 0^{th} interval is the setwise intersection of the i^{th} intervals. Because intervals are convex sets of points the intersection of a set of intervals will also be an interval. If there is inadequate information to find unique bindings for all variables, then `Intersect` fails.

For example:

`(Intersect (o 1 2 o) (c 1 2 c) (o 1 5 c) (c 0 2 o))` is true.
`(Intersect $x (c 1 5 o) (c 2 6 o))` is true
and binds $x to (c 2 5 o).

Intersect with Ports

Like `Meets`, `Intersect` can take port names to limit the possible variable instantiations to a finite set. The syntax of `Intersect` with port names is:

(`Intersect` <`interval`>$_0$ (<`port-name`>$_i$ <`interval`>$_i$)$^{\#}$))

If <`interval`>$_i$ is not completely specified by `Intersect`, then `Intersect` with ports assumes that <`interval`>$_i$ must be contained within an interval in the history of <`port-name`>$_i$.

The following is a partial description of an Or gate:

```
(if (and (intersect $t (in-1 $t1) (in-2 $t2))
         (ival (val (port in-1 or-gate) on) $t1)
         (ival (val (port in-2 or-gate) on) $t2))
    (ival (val (port out or-gate) on) $t))
```

If `in-1` having the value `on` causes the above rule to fire, then `Intersect` will do a lookup with 2 out of 3 variables unbound. Without the port restrictions there are an infinite number of possible bindings for `$t2`. With the requirement that `$t2` is an interval from `in-2`'s history, the number of possible bindings is limited to the number of interval-event pairs in `in-2`'s history. Because a port's history is finite, `Intersect` with ports is guaranteed to terminate with either a legal set of variable bindings, or if none exists, failure. `Intersect` without the port name restriction would suffer from the same infinite looping problem as `Meets`.

3.3 MARS Modifications

To reason with IRS predicates new MRS meta-level control rules have been added to MARS. The meta-level rules control how each of the IRS predicates are asserted and looked up. The version of MARS modified to handle intervals is called IMARS.

The output of a simulator is indicative of the model of persistence that it employs. MARS displays an event when the current simulator time reaches the point in time at which the event occurs. MARS stops displaying the event only when another event is asserted at the same port and replaces the first.

IMARS displays an event when the current simulator time reaches the lower bound of the interval corresponding to the event. IMARS stops displaying the event when the current simulator time reaches the upper bound of the interval. If no new event is asserted at that time, then IMARS indicates that the event at the corresponding port is `unknown`. Section 4.3.2 explores how permitting ports to have a null behavior permits a new type of reasoning.

If the upper bound of an interval over which an object's behavior is defined is `persists`, then IMARS acts in the same manner as MARS. The event is assumed to persist until a new behavior is asserted. MARS has an implicit model of persistence. An event is assumed to continue to occur until another event is asserted at the same place in the design. IMARS has an explicit model of persistence. An event only occurs during the interval at which it was asserted to occur. A port has the behavior `unknown` at some time if no event is explicitly asserted to exist at that time.

Lookups in the respective simulators are also different because of the change in time representation. When a lookup is done on a `true` statement, if nothing in the corresponding port's history is asserted at that point in time, then MARS looks to the previous assertion in the history. If the previous assertion unifies with the first argument of the `true` statement, then the MARS lookup returns the result of the unification.

Referring to the previous assertion in the history of the port is another manifestation of assumed persistence. If there is no behavior at the current point in time, then MARS assumes that the last behavior asserted is still true.

If an IMARS lookup fails to find an exact match between the second argument to `ival` and the time element of a port's history, IMARS uses the `contains` predicate to determine if an interval in the port's history contains the one being looked up. If one does, then the corresponding event entry in the port's history is unified with the first argument of the `ival` statement. The result of the unification is returned as the result of the lookup. This modification to the straightforward lookup captures the fact that the assertion that a port has a constant behavior over an interval implies that the port has the behavior at every subinterval of the interval.

In both MARS and IMARS there is a current simulator time, and consequently the concepts of past and future times. The current simulator time is either the point (MARS) or the lower bound (IMARS) of the assertion at the top of the simulation heap. In neither system may an assertion be made about a fact in the past, nor may a lookup be done on an event in the future. IMARS uses these restrictions to replace all `persists` markers occurring in intervals that start in the past with real number values.

If in the history of an object there exists an interval with an upper bound which is `persists` and following that interval is another interval, then when the current time is greater than the lower bound of the succeeding interval, the persists marker is replaced by a value such that the interval in which it occurs and the succeeding one meet. This can be done because no event can be asserted to exist between the two intervals; they are both in the past.

For example, the history of a port might contain the pairs 1 over the interval (o 2 persists c), and 0 over the interval (c 5 7 o). When the current time exceeds 5, there is no way a new event can occur between time 2 and time 5, hence `persists` is replaced with a 5. In addition, the `c` associated with the `persists` marker is replaced by an `o`.

Even before time 5, say at time 4, one has additional information about the persistence marker. At time 4 (o 2 persists c) can be replaced by (o 2 4 o) and (c 4 persists c). This replacement allows a lookup to use all of the history up to, and including the current time.

Every time an assertion or a lookup is made at a port, IMARS goes through the history of the port and removes all of the `persists` markers from the past.

3.4 An Example

Figure 3.1 illustrates a 74LS10 3 input positive Nand [The81] gate as it is modeled in Helios. `A`, `b`, and `c` are the input ports. `Y` is the output port. For illustration purposes the behavior of the Nand gate first is modeled using Corona and time points. Then, another rule is given in IRS with the time interval vocabulary. The first four assertions describe 3 input boolean nand.

```
Device nand3
  Ports:   A, C, B, Y
  No states

;;74LS10
;;3-Input Positive Nand-Gate
;;
;;Propagation Delay:     10ns
;;Setup time:            none

(bnand3  0 $x $y 1)
(bnand3 $x  0 $y 1)
(bnand3 $x $y  0 1)
(bnand3  1  1  1 0)

;;Point Model
(if (and (true (val (port a nand3) $a) $t)
         (true (val (port b nand3) $b) $t)
         (true (val (port c nand3) $c) $t)
         ($+$ 10 $t $t0)
         (bnand3 $a $b $c $y))
    (true (val (port y nand3) $y) $t0))

;;Interval Model
(if (and (intersect $sect (a $ival-a) (b $ival-b) (c $ival-c))
         (ival (val (port a nand3) $a) $ival-a)
         (ival (val (port b nand3) $b) $ival-b)
         (ival (val (port c nand3) $c) $ival-c)
         (idelay 10 $sect $ival-y)
         (bnand3 $a $b $c $y))
    (ival (val (port y nand3) $y) $ival-y))
```

Figure 3.1: **A Nand Gate**

3.4.1 MARS & The Point Model

As an example of how the point description in figure 3.1 might be used in a larger simulation, assume that when the current simulator time is 2, the event heap contains the following entries:

```
                  top
(true (val (port a nand3) 1) 2)
(true (val (port b nand3) 0) 2)
(true (val (port c nand3) 1) 2)
(true (val (port b nand3) 1) 4)
```

MARS begins by removing the top element from the heap. `(True (val (port a nand3) 1) 2)` matches the first conjunct in the first rule, so that rule is placed in a list of *fired* rules. `$A` and `$t` are bound to 1 and 2 respectively, and MARS looks up the second conjunct. There is no assertion that has been already made that matches the second conjunct so the rule fails. The event of `a` taking on the value 3 is placed in `a`'s history, and MARS removes the second element from the heap.

The first rule is again the one to fire. This time the second conjunct matches the event. 0 is bound to `$b` and and 2 to `$t`. MARS then goes back to the first conjunct in the rule, substitutes 2 for `$t`, and does a lookup on `(true (val (port a nand3) $a) 2)`. This time there is a match because of the first assertion; \$a is bound to 1. MARS continues with the third conjunct because the second one is the one that caused the rule to fire. The third rule does not have a match so the rule fails. The second event is placed in port `b`'s history, and MARS continues with the third event.

The third event also causes the first rule to fire. `$C` is bound to 1, and `$t` is bound to 2. MARS goes back to the first conjunct, does a lookup, and binds `$a` to 1. In the same manner `$b` is bound to 0. The fourth conjunct is the `+` predicate. The first argument is 10, the second is bound to 2, consequently `$t0` is bound to 12. This conjunct is included to model a 10ns delay between when an event occurs on the inputs of the device, and when the effect is seen on the output.

The last conjunct matches the second `bnand3` assertion, subsequently $y is bound to 1. In the consequence of the rule there are two variables, $y which is bound to 1, and `$t0`, which is bound to 12, consequently `(true (val (port y nand3) 1) 12)` is asserted and placed in the event heap.

The current time is now 4, and the two events in the heap are:

```
              top
 (true (val (port b nand3) 1) 4)
(true (val (port y nand3) 1) 12)
```

The top rule is removed and again the first rule is fired. This time `$b` is bound to 1 and `$t` is bound to 4. MARS goes to the first conjunct and does a lookup on `(true (val (port a nand3) $a) 4)`. There is no event in `a`'s history at time 4 so MARS assumes that the last event to have occurred before time 4 persisted. As a result, `(true (val (port a nand3) 1) 4)` is assumed to be true and $a is bound to 1. `$c` is bound to 1 in the same manner.

After all of the conjuncts have been evaluated, `$a`, `$b`, and `$c` are all bound to 1, and $y is bound to 0. The consequence which is placed in the event heap is `(true (val (port y nand3) 0) 14)`. The remaining two events in the heap do not match a conjunct in any rule, and consequently do not cause any rule to fire. They do, however, cause the user interface to display the fact that port `out` gets the value 1 at time 12, and 0 at time 14. At time 14 the event heap is empty, so the simulation is over.

3.4.2 IMARS & The Interval Model

IMARS follows the same basic algorithm as MARS: Events cause rules to fire, and IMARS does lookups on the conjuncts of an antecedent to bind variables. For this example, assume that the initial state of the event heap is:

```
                    top
(ival (val (port a nand3) 1) (c 2 6 o))
(ival (val (port b nand3) 0) (c 2 5 o))
(ival (val (port c nand3) 1) (c 2 6 o))
(ival (val (port b nand3) 1) (o 5 7 o))
```

Because the assertions are ordered by their lower bounds this is one of the 6 possible states that the event heap could be in. When the first event is removed, it matches the second conjunct in the second rule, and causes that rule to fire. IMARS binds `$a` to 1 and `$ival-a` to `(c 2 6 o)`. IMARS then moves to the first conjunct and does a lookup.

Only one of the four variables, `$ival-a`, in the first conjunct is bound. To find bindings for `$ival-b` and `$ival-c` IMARS uses the information that `$ival-b` must come from the history of port `b`, and`$ival-c` from port `c`. Notice that the orts in the `Intersect` statement enforce constraints that will be imposed by the third and fourth conjuncts of the rule. No new constraints are added to the rule.

Because neither ports `b` nor `c` have any bindings in their histories the rule fails. Consequently, the initial event, `(ival (val (port a nand3) 1) (c 2 6 o))` is placed in port a's history, and IMARS goes on to the next event in the heap.

The consequences of asserting `(ival (val (port b nand3) 0) (c 2 5 0))` are analogous to those for asserting the first event.

When the next event from the heap is asserted, all of the other ports have a value. `(Ival (val (port c nand3) 1) (c 2 6 o))` matches the fourth conjunct in the interval model rule. `$c` is bound to 1, and `$ival-c` to `(c 2 6 o)`. IMARS continues by looking up the first conjunct in the rule.

Of the four variables, the only one in the first conjunct that is bound is `$ival-c`, to `(c 2 6 o)`. The next variable that needs to be bound is either `$ival-a` or `$ival-b`. `$Ival-a` is constrained in two ways. First, it must be an element of port `a`'s history. Second, it must have a non-empty intersection with `(c 2 6 o)`. The sole interval in port `a`'s history satisfies both of these restrictions, so $ival-a is bound to `(c 2 6 o)`.

The same two restrictions hold for the lookup of `$ival-b`, except the interval which `$ival-b` must intersect is `(c 2 5 o)`, the intersection of intervals `$ival-a` and `$ival-c`. The reason that the restricting interval can be smaller is that the `Intersect` predicate only returns true when all of its arguments intersect in a non-empty interval. The reason it is desirable that the restricting interval is smaller is that this further limits the domain over which the search for possible matches is made.

The final binding for `$sect`, the intersection of the times at which the different ports have values, is (c 2 5 o). After returning the bindings for the variables in `intersect`, IMARS looks up the values at each of the ports using these bindings.

Next, `IDelay` is used to compute the interval which is 10 time units after `$sect`. This models the 74LS10's 10ns delay. The last lookup done in the antecedent of the rule is the same as was done when time points were used to model the Nand gate, `$y` is bound to `1`.

The two variables in the consequence of the rule, `$y`, and `$ival-y`, are bound to `1` and `(c 12 15 o)` respectively so the result of the rule firing is that `(ival (val (port y nand3) 0) (o 15 16 o))` is placed in the heap. At this point the current simulation time is 5, and the state of the event heap is:

```
                    top
 (ival (val (port b nand3) 1) (o 5 7 o))
(ival (val (port b nand3) 0) (c 12 15 o))
```

The top rule is removed from the heap, and the interval model rule again is the one fired. This time, as a result of a lookup on `intersect`, `$sect` is bound to `(o 5 6 o)`, the intersection of `(o 5 7 o)`, and `(c 2 6 o)`. `IDelay` delays the overlapping interval by 10 time units, `bnand3` computes the 3-input boolean nand of the inputs, and the resulting asserted expression is, `(ival (val (port y nand3) 0) (o 15 16 o))`.

Neither of the remaining two statements in the heap cause any of the rules to fire. They do however, update the database to first indicate a value of 1 on the output port, then a value of 0, and finally a value of `unknown`. They will also be inserted into port `y`'s history to describe its behavior from time 12 to time 16.

3.4.3 Comparison & Contrast

In many ways the two models of the 74LS10 are similar. Both are fired by changes of the values of the input ports, and both use the input port values, via the `bnand3` predicate, to compute the output. Also, in both cases, a predicate was use to compute a delay between when the change of input values occurs, and when the change of value of the output is asserted.

The differences between the two models are also apparent. With the point model of time, when there is no explicit entry in the database for the value on the input ports at time 4, the value that was asserted to exist at time 2 is assumed to persist. Under the interval model, no assumption of persistence was made. Only explicit database entries matched the conjuncts in the rules.

Under the interval model the values did not remain on the ports forever. MARS ended with each of the inputs having the value 1, and the output having the value 0. At time 7, IMARS set the values of the inputs to `unknown`. At time 16 the value of the output was set to `unknown`.

The interval time model's rejection of the implicit persistence assumption created the need for a new predicate which computes when all of the relevant ports whose values are being looked up, do not have the value `Unknown`. This need was satisfied by the `Intersect` predicate. `Intersect` computed the interval over which all of the input ports to the Nand gate had a known value. Only during this interval could a valid output be computed.

The change in representation of time also created the need for a new predicate to compute delay. Previously, under the point model, `+` was sufficient to increase the real number value representing the point. Intervals needed an operator which would increment both endpoints of the interval and preserve the boundary types (closed or open). `IDelay` satisfies this requirement.

In the next chapter I demonstrate that as a result of the aforementioned differences the interval model of time is a more powerful representation. Certain reasoning tasks require explicit control of the persistence of values at a port. Other tasks need to know when a port does not have an explicitly stated value. Only the interval model of time offers control required to perform these tasks.

Chapter 4

Examples

IRS introduces both a vocabulary for explicitly denoting intervals of time, and a set of predicates for manipulating intervals of time. A question which must be asked when one introduces a new vocabulary is, "What is the relationship between the vocabulary, and the situations or plans which can be represented using that vocabulary?" More simply, "What is the representational power of the language?" This chapter breaks these questions down into three categories, and for each category shows, via examples, that IRS's new vocabulary increases Helios's representational power and ability to reason about designs.

The format of each example is to first discuss the desired behavior, next give the point model of the behavior and describe its inadequacies, and then show how the interval model overcomes the inadequacies. The following three questions are used to categorize the examples:

1. Does the new vocabulary increase the user's ability to correctly specify the behavior of the artifact being designed?
2. Does the new vocabulary allow any design related tasks to be performed more efficiently?

3. Are there tasks which may be performed with the new vocabulary that can not be performed without it?

4.1 Correctness

It is impractical, if not impossible, to model perfectly an artifact being designed. Digital designs are themselves an abstraction of the true analog behavior of a device. The questions arise: To what degree of accuracy does one want to specify the behavior of a device? Is such a degree of accuracy achievable with the available description vocabulary?

The following three examples demonstrate how Corona's lack of representational power limits the correctness with which one is able to model digital designs. In the first case, the point model is unable to model setup time because there is no way of specifying that a port's behavior is unchanging at an infinity of points. The second case demonstrates that the point model cannot capture the concept of two events meeting and consequently cannot describe consecutive behavior. The last example shows that to model tristate devices the designer is forced to emulate intervals with points.

4.1.1 Setup Time

The setup time associated with an input to a clocked device is, "the time required for the input data to settle in before the triggering edge of the clock" [Fle80]. As described here, setup time is not a point, but an interval. Requirements made of an input's setup time must hold over the whole interval.

The currently used technique for modeling setup time in Corona is to require that the value on the input be the same at the beginning and end of the setup time.

The latch described in figure 4.1 has a 20ns setup time associated with

```
Device latch
  Ports:   D, G, Q
  No states

;;Width of enabling pulse:      Min 20ns
;;Setup time:                   Min 20ns
;;Propagation Delay:            Typ 10ns
;;  The TTL Data Book, p. 7-39

;;Point model
;;Persistence assumed
(if (and  (+* $t0 ($t1 30) ($t 10))
          (true (val (port D latch) $d) $t)
          (known (true (val (port D latch) $d) $t1))
          (true (val (port G latch) 1) $t)
          (known (true (val (port G latch) 1) $t1)))
    (true (val (port Q latch) $d) $t0))
```

Figure 4.1: **Point Model of a Latch with a 20ns Setup Time**

its D and G inputs. D's setup time is enforced by the 2nd and 3rd conjuncts in the rule's antecedent. When a statement about the value of D is made, it matches the 2nd conjunct, causes the latch rule to fire, and binds $d to a specific value. The third conjunct will block Q's value from being updated, *i.e.* the rule from succeeding, if D's value was not the same 20 time units previous to the current time.

The known statement wrapped around the 3rd conjunct blocks MARS from causing the rule to fire because a true statement matches the conjunct. Because MARS requires that all statements in the antecedent of a rule occur at or before the current simulation time, a rule can not fire due to a conjunct whose time stamp is not the maximum of all of the conjuncts' time stamps. By wrapping known around the 3rd conjunct, it saves MARS from firing the rule and finding out that the second conjunct is making a query about 20 time units in the future. Known is included to increase MARS's efficiency, but otherwise does not affect the meaning of the rule.

There are two inadequacies with this model setup time. Recall that MARS only fires a rule when one of its conjuncts appears at the top of the scheduling heap. When the data port of the latch first attains a new value, it obviously will not satisfy the setup time requirement. At the end of the setup time, however, there is nothing in the scheduling heap to cause the rule to fire.

The first problem with the MARS modeling methodology results in a rule not firing when it should. The second problem results in MARS not blocking a rule from firing when the setup time requirement is not satisfied. Notice that the setup time requirement is modeled as a constraint only upon its starting and ending points. That is, the setup time requirement is considered to be fulfilled if a port has the same value at the beginning and end of the setup interval. No constraint is placed on the intermediate times. A short deviation of value (a glitch) will not be detected in the value of the data port.

Specifically, consider port D of latch. If D has the value x at time 0, the value y at time 10, and again attains the value x at time 20, then the model of the setup time requirement has been satisfied even though the value on the data port had not had enough time to settle.

Solutions to each of these problems have been proposed and/or implemented. To handle the scheduling problem, MARS includes meta-level rules for inserting a special marker into the scheduling heap. The marker signals that a certain rule should be checked, even though none of its conjuncts are being asserted.

This patch is not satisfying in the sense that it is not consistent with the MARS paradigm: The rule associated with a device only needs to be checked when the input values of the device change. Furthermore, this fix can lead to an indeterminate number of checks being placed in the scheduling heap. The efficiency problems with this patch and how an interval vocabulary solves them are discussed in section 4.2.2 of this chapter.

One way of viewing the point model of setup time is as a sampler with frequency 2. Using this model, glitches can occur because the sampling frequency is not high enough. The designer must choose a granularity and sample at a high enough rate to detect glitches that are larger than the prespecified granularity.

For example, instead of modeling a 20ns setup time by requiring that the value on the data port be the same at time `$t`, and at `$t-20`, the designer chooses a granularity of, say 5ns, and samples the value of the port at times `$t-20`, `$t-15`, `$t-10`, `$t-5`, and `$t`.

This solution is inadequate first because of its bulkiness. A 20ns setup time is relatively small. To detect 2ns glitches, *each* data port would need 11 conjuncts to model its setup time. More important, the solution is inadequate because, for any granularity that the designer chooses to sample at, a shorter length glitch can go undetected.

Another proposed solution to the glitch problem is to associate a state variable with every port, and store the last time a change was made to the value of that port in the state variable. With this mechanism, enforcing setup time is equivalent to requiring that the value (<current-time> − <last-update-time>) is greater than the minimum setup time.

Again, the problem of efficiency arises with this solution. It requires a new state variable to be associated with each port in the design. Furthermore, if this methodology is adopted in MARS, the number of rules required to describe a design doubles. Everytime a value is asserted to exist at a port, another rule must assert that <last-update-time> holds the current simulator time.

Better than patching the point model of time is to have a vocabulary with primitives that capture the concept of setup time. The interval is that primitive. Figure 4.2 describes the latch from figure 4.1 using the interval model. Setup time is satisfied if the length of the interval representing it is large enough. The sampling frequency of the interval is infinite because an assertion made about an interval of time is equivalent to the assertion made at every point during the interval. The *last-update-time* is analogous to the low bound of the interval.

Each of the patches to the point model is an attempt to incorporate some part of the interval model into MARS. The interval model concepts that each of the patches capture are themselves a description of what vocabulary one needs to correctly specify setup time in a design.

4.1.2 Consecutiveness

Not all digital behavior results from constant values existing on the inputs of devices. For example, the behavior of a D flip-flop is such that, the value of the `Q` output port is set to the value of the `D` input port when the `clk` input is rising, *i.e.*, when the clock input was 0 a *moment* before some point in time, and is 1 at the point in time.

In the case of the D flip-flop, the behavior of the `Q` output of the flip-flop is a function of a change in the behavior of the `D` input, *i.e.*, it switching from 0 to 1. The method used to model the change using time points resembles the method used to model setup time. When the clock input of the flip-flop becomes 1, an assertion matches the second conjunct of the flip-flop rule, and the rule is fired. The third conjunct of the rule checks

```
;;Interval model
;;Persistence must be modeled

(if (and (intersect $sect (D $d-ival) (G $g-ival))
         (ival (val (port D latch) $d) $d-ival)
         (ival (val (port G latch) $g) $g-ival)
         (ilength $len-d $d-ival)
         (ilength $len-g $g-ival)
         (> $len-d 20)
         (> $len-g 20)
         (idelay 30 $sect ($lo-b $lo $hi $hi-b)))
    (ival (val (port Q latch) $d) (c $lo persists c)))

(if (and (intersect $sect (G $g-ival) (Q $q-ival))
         (ival (val (port G latch) 0) $g-ival)
         (ival (val (port Q latch) $q) $q-ival))
    (ival (val (port Q latch) $q) $sect))
    (true (val (port Q latch) $d) $t0))
```

Figure 4.2: **Interval Model of a Latch with a 20ns Setup Time**

```
Device dff
  Ports:   D, CLK, Q
  No states

;; Modified SN74LS377 Octal D-type flip-flop
;;
;; Data Setup Time:       20 ns
;; Propagation Delay:     18 ns

;;Point Model:
;;Persistence assumed
(if (and (+* $t ($t0 1) ($t1 20) ($t2 -18))
         (true (val (port clk dff) 1) $t)
         (known (true (val (port clk dff) 0) $t0))
         (true (val (port d dff) $d) $t)
         (true (val (port d dff) $d) $t1))
    (true (val (port q dff) $d) $t2))
    (ival (val (port q dff) $d) $q-ival))
```

Figure 4.3: **Point Model of an Edge-Triggered D Flip-Flop**

to see that the value of the clock was 0, one time unit previous to when the value was 1.

Because a point model of time is used to model the rising edge, the design engineer is forced to choose a specific length of time, and check to see that the clock was 0 that length of time previous to when the clock value becomes 1. As in the setup time example, the choice of time length represents the design engineer's choice of design description granularity.

In figure 4.3 the granularity of description is 1 time unit. If the `clk` input of the D flip-flop is driven with a period 1 clock, then the D flip-flop rule will never fire. Whenever the clock input has a certain value, it will have had the same value 1 time unit before.

A solution to this problem is to reduce the granularity used to describe the flip-flop. If the D flip-flop rule checked the value of the clock input one half time unit before the input becomes 1, then the rising edges of the clock would cause the rule to fire. This solution, however, requires the design engineer to anticipate the minimum width of any stray glitches that could occur at the clock port of the flip-flop and does not solve the problem in the general case. A glitch can occur at a finer level of granularity than that which the rule employs for detecting a rising edge.

The underlying problem in describing edges is not one of granularity, but of the inability of the point model of time to represent the *consecutiveness* of events. A rising edge is a combination of two consecutive events. Specifically, a rising edge occurs at a port when the value of the port is 0, followed immediately by 1. With a vocabulary for intervals one is able to express the concept of consecutiveness, or abutment; unlike points, intervals can meet.

The two rules in figure 4.4 give the interval model of the flip-flop's behavior. The first rule differs from the point model in three ways. First, the setup time is modeled as an interval. This is done as in the previous section. Second, in the point model of time, the propagation delay is accounted for by adding 18 time units to the time when rise occurs. Using the interval model, the `idelay` predicate is used to slide the interval up by 18 time units.

```
Device dff
  Ports:   D, CLK, Q
  No states

;; Modified SN74LS377 Octal D-type flip-flop
;;
;; Data Setup Time:        20 ns
;; Propagation Delay:      18 ns

;;Interval Model:
;;Persistence must be modeled.

(if (and (meets $c (clk $lo-time) (clk $hi-time))
         (known (ival (val (port clk dff) 0) $lo-time))
         (ival (val (port clk dff) 1) $hi-time)
         (ival (val (port d dff) $dat) ($a $b $c $d))
         (ilength $d-len ($a $b $c $d))
         (< 20 $d-len)
         (idelay 18 $hi-time $q-time))
    (ival (val (port q dff) $dat) $q-time))

;;Persistence
(if (and (ival (val (port q dff) $d) ($a $x $x $b))
         (ival (val (port clk dff) $c) ($w $x $y $z))
         (idelay 18 ($w $x $y $z) $q-ival))
    (ival (val (port q dff) $d) $q-ival))
```

Figure 4.4: **Interval Model of an Edge-Triggered D Flip-Flop**

The important difference is in how the edge is detected. Under the point model an edge is modeled as the value being 1 at time `$t`, and 0 at time `$t0`. Under the interval model of time, a rising edge occurs when the value of the port is asserted to be 1 over an interval, `$hi-time`, the value was 0 over an interval `$lo-time`, and the two intervals meet.

The second rule is necessary because persistence is not assumed under the interval model. We must model the fact that the output of a D flip-flop does not change if the edge is not rising.

4.1.3 Tristate Devices

A common situation in digital design is output from several devices being tied together by a single wire called a bus. All of the output must have the same value, otherwise the value of the bus is in a state of *contention*. To avoid the problem of contention *tristate* devices are used. Tristate devices are able to disable their output, cutting themselves off from the bus. If a device on the bus asserts that it has a value and the rest of the devices are disabled, then there is no contention.

The point model of a disabled output includes two patches to MARS. First, a reserved value, `hi-z`, is asserted as the value of an output port when it is disabled. Second, special wire models are used to connect tristate devices. The special wires are equivalent to devices that collect all of the non-`hi-z` inputs and if they are the same asserts their value at the output. If all of the inputs to the wire are `hi-z`, then so is the output.

There are several reasons why the patches to MARS are undesirable. The use of reserved value names that have special meaning is generally discouraged. Errors that result from a design that unknowingly uses a reserved variable name are difficult to track down.

Having two different types of wires is also an extremely *ad hoc* way of modeling tristate devices. It violates modularity principles by coupling the behavior of a wire and the device it is connected to.

The introduction of `hi-z` into MARS's vocabulary is, in effect, an attempt to model intervals. `Hi-z` is equivalent to the upperbound of an interval. Asserting that a port has the value, `hi-z` is asserting that it no longer has a value. The concept of not having a value is captured by `hi-z`'s unique behavior of not overwriting other values.

IMARS captures the desired behavior of `hi-z` in a single well-defined design model. Instead of special tristate wires detecting contention, IMARS is able to detect contention at all ports. This capability is discussed more in section 4.3.1 of this chapter.

Because IMARS does not have an implicit model of persistence, it includes the concept of a port not having a value. A gap between the intervals when a port has a value, is an interval during which the port has no value.

To model tristate devices and a tristate bus nothing special need be added to IMARS or IRS. Contention exists when two devices that are connected are outputting different values onto the bus. Devices with disabled outputs do not assert any value onto the bus.

4.2 Efficiency

Part of the question, "What is the representational power of a new vocabulary?" is declarative: What situations can be represented? Another facet of representational power is, how efficiently may tasks be performed with the representation?

The previous section presented examples which demonstrate that time intervals allow a broader class of situations to be modeled correctly. The examples in this section demonstrate that certain tasks may be performed more efficiently using the interval model.

4.2.1 Latch

A forward chaining task is one in which conditions cause rules to fire, which in turn setup new conditions, which cause new rules to fire. The examples presented thus far have demonstrated problems that can occur with a forward reasoning simulator. Either a rule fires when it should not, or a rule does not fire when it should.

Different types of problems can occur with backward reasoning tasks. Backward reasoning tasks are characterized by being given a set of existing conditions and, as a goal, trying to determine which rules fired to cause the conditions to exist. Diagnosis and test vector generation are two tasks that involve backward reasoning.

A problem associated with backward reasoning is that of infinite regress. Smith gives an example of this over the domain of integers [Smi85]. After presenting his example I will show that the problems are a result of the discreteness of integers and that the map directly onto the point model of time. Smith's example:

```
We are given the rules:
        PosInteger(1), and
        PosInteger(X) ← PosInteger(X-1).
```

Proving that 3 is a positive integer is a "straight backward" process. 3 is a positive integer if 2 is, 2 is a positive integer if 1 is, and we know that 1 is a positive integer. When the same question is asked of 2.5, the problem regresses infinitely. 2.5 is an integer if 1.5 is. 1.5 is integer if .5 is, and so on.

As a solution, Smith proposes representing what is contained in the data base, and using that knowledge to cut off infinite regressions. In the above discussion, 1 is the smallest integer in the database, if the argument to `PosInteger` ever falls below 1, then the rule fails.

An alternate solution is to use intervals and negation to assert when the `PosInteger` rule should fail. $\neg$`PosInteger`$([o, 0, 1, o])$ could represent

```
Device state-mach
  Ports:   clk
  States:  curr-state

(if (and (+* $t (1 $t1))
         (true (val (state curr-state state-mach) $s) $t1)
         (state-table $S $next-s)
         (true (val (port clk state-mach) 0) $t1) ;clk-rise
         (true (val (port clk state-mach) 1) $t)) ;clk-rise
    (true (val (state curr-state state-mach) $next-s) $t))

(state-table S0 S1)
(state-table S1 S0)
```

Figure 4.5: **Point Model of a Two State Machine**

the fact that, if the argument to PosInteger is between 0 and 1, exclusive, then the deduction is false and should stop.

With this mechanism in place, the query PosInteger(2.5) reduces to PosInteger(1.5). And PosInteger(1.5) reduces to PosInteger(.5), which in turn would cause the query to return False.

Infinite regression also can occur with backward reasoning tasks in the digital design domain. The device in figure 4.5 is a clock driven, two state machine. On a rising edge of the clock input the state machine moves to the next state in the state table. In this example, there are only two states, S0 and S1.

A desirable subtask of diagnosis and test vector generation is to determine, given a set of conditions, a series of inputs and the times at which they occur that will cause the state machine to be in a certain state at a certain time. Using the point model, the conditions are specified by the true statement. As an example, consider the goal of determining the initial value of curr-state given the conditions:

```
(true (val (port clk clock) 0) 0), and
(true (val (state curr-state state-mach) S1) 15)
```

Also, assume that `clk` is driven by a clock:

```
(if (and (true (val (port clk clock) $x) $t))
         (bnot $x $y)
         (+ $t 5 $t1))
    (true (val (port clk clock) $y) $t1))
```

Working backward through the rules that model the state machine, `$t` is bound to 15, `$t1` to 1, and `$next-s` to `S1`. New goals are setup, `clk` must be 1 at time 15 and 0, at time 14. `$S` is bound to `S0`, and the value of `curr-state` at time 14 must be `S0`.

Because `clk` must be 1 at time 15, working backward through the clock behavioral rule, it must be 0 at time 10, 1 at time 5, and 0 at time 0. Carrying out the same regression with the rest of the conditions, the original goal is satisfiable if `curr-state` is `S0` at time 0.

Like the `PosInteger` example, if the time at which the goal state is supposed to exist is not an even multiple of the clock frequency, then the backward reasoning facility regresses infinitely. If the goal were, "`curr-state` is `S1` at time 8," a subgoal would be setup that it is `S0` at time 3. There is no fact in the state machine's history to match this, so it checks at time -2, and so on.

Smith's solution [Smi85] is to add a meta-level rule that detects when the argument to the query is less than the earliest time that an assertion is made about `curr-state`'s value. If the goal time were very large with respect to the clock increments, however, then the number of checks that would need to be made would also be large. Furthermore, the addition of the meta-level rule to solve this is too problem specific. A common way of asserting that a port has always had a value is to assert that it received the value at time $-\infty$, and let the value persist. In this instance, Smith's least value solution would never reach the base case.

```
Device state-mach
  Ports:   clk
  States:  curr-state

(if (and (meets $meet-pt (clk $ilo) (clk $ihi))
         (ival (val (port clk state-mach) 0) $ilo)
         (ival (val (port clk state-mach) 1) $ihi)
         (ival (val (state curr-state state-mach) $s)
       ($x $y $meet-pt $z))
         (state-table $s $next-s))
    (ival (val (state curr-state state-mach) $next-s) $ihi))

(if (and (meets $meet-pt (clk $ihi) (clk $ilo))
         (ival (val (port clk state-mach) 0) $ilo)
         (ival (val (port clk state-mach) 1) $ihi)
         (ival (val (state curr-state state-mach) $s)
       ($x $y $meet-pt $z)))
    (ival (val (state curr-state state-mach) $s) $ilo))

(state-table S0 S1)
(state-table S1 S0)
```

Figure 4.6: **Interval Model of a Two State Machine**

The interval solution proposed for the integer rendition of the problem also works in the digital domain. The interval description of `state-mach`'s behavior is given in figure 4.6. The behavior is captured by two rules instead of one: First, the value that a state machine receives at the rising edge of a clock stays with it during the interval the clock value remains high. The second rule says that the current state of the state machine does not change when the clock edge falls, and that the value remains constant as long as the value of the clock is 0.

The second rule is a frame axiom [Hay73], that explicitly specifies how long the value of `curr-state` persists. As a result, the behavior of the state machine is completely specified. There is never a time at which it does not have an explicitly asserted behavior. Furthermore, because the behavior of the component is completely defined, the backward reasoning regression is not infinite.

In the interval model the initial conditions and the clock behavior are given by:

```
(ival (val (port clk clock) 0) (c 0 5 o))
(ival (val (state curr-state state-mach) S0) (c $x 8 o))


(if (and (ival (val (port clk clock) $x) $i))
         (bnot $x $y)
         (idelay $i 5 $i1))
    (ival (val (port clk clock) $y) $i1))
```

There are two possible paths that the backward reasoner can start down to achieve `S0` at time 8. They correspond to the two rules that model the state machine. If the clock ends up being high at time 8 then the first rule is the correct path, otherwise the second.

Following the first rule, 8 must be contained in the interval `$hi`. `$hi` meets an interval `$lo` during which `clk` was 0. `$lo` matches the fact, `(ival (val (port clk state-mach) 0) (c 0 5 o))`. Back substituting, this constrains `clk` to 1. Finally the test generator concludes that the `curr-state` must have been `S1` at time 0.

The previous example was generated manually. No modifications were made to the Helios test vector generator or diagnostician to perform such reasoning. Achiever [Joy83] performed a restricted case of the backward reasoning described here. The results on the single example of a D flip-flop show an order of magnitude increase in the time efficiency of the backward reasoner.

```
Device Same
  Ports:   A, Y
  No states

(if (and (+ $t 10 $t1)
         (true (val (port A same) $x) $t)
         (true (val (port A same) $x) $t1))
    (true (val (port Y same) 1) $t1))
```

Figure 4.7: **Point Model of Device Requiring a Recheck Marker**

4.2.2 Recheck

In the setup time discussion it was shown that because MARS only checks a rule when an assertion from the scheduling heap matches a conjunct in the rule, it is possible that the rule does not fire even though the antecedent of the rule is satisfied. This problem occurs whenever two conjuncts in the antecedent of a rule reference the same port at different times.

The device, `same` in figure 4.7 exemplifies the problem. Consider the MARS simulation started with the assertion:

```
(true (val (port A same) 0) 0)
```

MARS removes the assertion from the heap. It first tries to bind `$t` to 0. This results in `$t1` being bound to 10, and in the rule failing because a lookup was attempted on a statement with a time stamp greater than the current simulator time. Next, MARS matches the removed statement to the third conjunct of the rule. `$t1` is bound to 0, `$t` is bound to -10 and the rule fails because `A` did not have a value at time -10. No more assertions are in the scheduling heap so the the simulation is over without the `same` rule firing.

Assuming persistence, at time 10 `same`'s antecedent is satisfied. `Y` should have a value after time 10. To correct this problem, whenever a rule fails because a conjunct's instantiated time stamp is in the future, a marker is placed in the heap at the time that the conjunct should be checked. The marker matches the failed conjunct, and consequently causes the rule to refire.

In the above case, the marker has the form `(true (val (port A same) $x) 10)`. The effect of the marker is the same as if there was a rule that found the value of `A` at time 10 and reasserted it. When the marker matches the second conjunct, `$t1` is bound to 10, `$t` is bound to 0, and the rule succeeds.

In addition to the marker matching the second conjunct, it matches the first. When it matches the first, `$t` is bound to 10, `$t1` to 20, and again a marker is placed in the heap because a lookup was done in the future. Once placed into the queue, the marker will reassert itself every `n` time units where `n` is the difference between the time stamps of the conjuncts that check the same port. The combinatorics of the reassertion scheme can overburden MARS for many designs.

The important difference between the point and interval models is captured not only in the behavioral descriptions; the representation of simulation data is equally important. The interval model simulation data is:

```
(ival (val (port A same) 0) (c 0 <upper-bound> o))
```

`Same`'s output is 1 during the interval that results from intersecting `(c 0 <upper-bound> o)` delayed by 10 time units with itself. In IMARS there is no implicit assumption of persistence, and consequently no need for the recheck marker. Assertions include the whole interval over which a port has a value. Waiting for the current simulator time to change will not increase the upper bound of the interval.

4.3 New Tasks

The representational power of a language can be discussed in terms of its declarative, and procedural aspects. The declarative aspects of a language encompass the question: What can be said with this language? The procedural aspects are captured in: What tasks may be performed with this language?

In section 4.1 of this chapter, I presented examples which demonstrate that the declarative power of the point model of time and its associated vocabulary is less than that of the interval model of time. In this section, I demonstrate the procedural power of the interval model of time: What new tasks may be performed with the interval model of time that can not be performed with the point model?

4.3.1 Contention

In the MARS algorithm, as it is described in chapter 3, no ordering is specified in the scheduling heap for assertions with the same time stamp. Because MARS employs a depth first, forward chaining inference mechanism [Sin83], the actual ordering is that the last statement, out of a set of statements made about the same point in time, is the one closest to the top of the heap. In the case that two statements assert that a port has different values at the same point in time,"...the old value is removed. There is an implicit assumption that the latest information is more correct..." [Sin83, p.21].

Singh gives the example in figure 4.8 to support the need for overwriting. A change of the value at point `a` in the circuit causes three sets of rules to fire: `I`, the rules associated with the inverter, A_b the rules associated with the And gate that fired because of a change at point `b` in the circuit, and A_c, the rules associated with the And gate that fired because of a change in the value at point `c`, in the circuit.

The only constraint on the firing order of these rules is that `I` must precede A_b. This leaves three possible orderings: (`I`, A_b, A_c), (`I`, A_c, A_b)

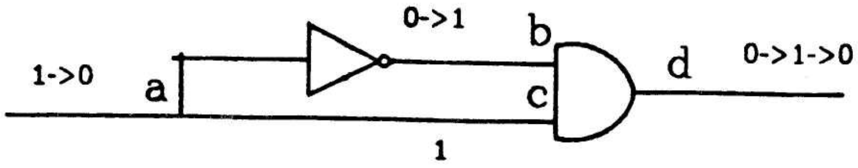

Figure 4.8: **Simulation of Zero Delay Components**

and, (A_c, I, A_b). Because there is no delay associated with any device, the results of all rule firings are assertions at the point in time that the original change at a occurred.

Consider the firing order (I, A_b, A_c) when the value at point a is changed from 1 to 0. Point a going from 1 to 0, causes the value of b to go from 0 to 1, which causes the output of the And gate to have the value 1. Following the lower path, a going from 1 to 0 causes c to change from 1 to 0, causes the output of the And gate to have the value 0. Two conflicting assertions have been made about the value of the And gate at a single point in time.

As per the overwrite rule, the last assertion about the value at d takes precedence. This models the correct behavior. However, representational power is sacrificed by implicitly assuming that the last value asserted is the correct one. With overwriting allowed, it is impossible to detect a certain type of incorrect behavior specification —those in which the design engineer specifies more than one behavior for a port at a single point in time.

At the root of the need for overwriting is MARS's implicit assumption of persistence. From the example, the first value asserted at point d was a result of firing the `And gate` rules with the new value at `b`, and the value that persisted at `c`. The conflicting assertion about the value at `d` was made as a result of using the new value at `c` and the new value at `b`.

If MARS did not assume persistence, the first And gate rule firing, A_b would not occur because the `c` port of the And gate would not have a value until the value was propagated from `a` to `c`.

IMARS does not assume persistence, and consequently does not require overwrite rules. By doing away with the need for overwriting, IMARS can consider any attempts at overwriting to be errors in the design specification and report them as such. Either the design engineer erred —two different behaviors were specified to exist at the same place and time, or a contention condition exists. Two devices are trying to drive a third with different values. The ability to differentiate between inconsistency in design specification and normal simulator behavior is an important task that IMARS can do because of the introduction of intervals into the design vocabulary.

4.3.2 No Behavior

Apart from tristate devices, it is usually the case that an accurate model of a device specifies a behavior of the device at all times. An indication of a possibly faulty model is the lack of behavior at an output port during some interval of time.

One would like to be able to detect when, during the simulation of a device, the behavior of the device is unspecified. Using the point model of time this is extremely difficult; MARS implicitly assumes the persistence of values at ports. Consequently, once a port attains its initial value, it will never again not have a value.

Because IMARS does not have an implicit model of persistence, gaps of no behavior can appear in port histories. These periods of no behavior indicate that either the behavior of the device's inputs was not specified, or the behavior of the inputs was specified but none of the rule antecedents were satisfied by those inputs.

To find the periods of no behavior, it is sufficient to scan down the history of each port checking whether successive intervals meet. If they do not, then there is a interval of time during which a port had no behavior. Given this information the design engineer can decide whether the incompleteness of definition was intended, as in the case of tristate devices, or needs to be corrected.

4.4 Subsumption

I have shown that in many cases there are modeling tasks that cannot be performed using a point model of time, but can be performed with an interval model. The question arises: Are there tasks that can be performed with the point model that cannot be performed with the interval model? The answer is, no. The interval model of time subsumes the point model. This is to say, any statement made using the point model of time and its associated predicates can also be made using the interval model. The transformation from the point to the interval model is performed by the MRS meta-level rule:

```
(If (true (val (port $x $y) $z) $t)
    (ival (val (port $x $y) $z) (c $t persists c)))
```

If one wanted to omit the persistence assumption from the point model, then the rule would be:

```
(If (true (val (port $x $y) $z) $t)
    (ival (val (port $x $y) $z) (c $t $t c)))
```

The purpose of including rules like these is to show that no power or simplicity of description is lost in switching from the point to interval model. It is extremely important to remember that the goal of IRS is not to be equivalent to a point model. I have demonstrated that the point model of time is inadequate for modeling digital designs, and have showed that the explicit control of persistence afforded by the interval model solves the inadequacy.

Chapter 5

Summary & Conclusion

Allen introduced an interval-based temporal logic for describing temporal intervals as primitives in a knowledge representation language [All83]. I implemented parts of Allen's logic in IRS, and applied IRS to the problem of modeling digital designs. This thesis investigated the question: Of what use are temporal interval primitives for modeling digital designs?

Previous to my work, the Helios system (and more specifically its design description language, Corona) used a point model of time. The behavior of a design was described as a set of events occurring at points in time. When no event occurred the behavior of the previous event was assumed to persist.

By introducing IRS and its associated interval vocabulary I was able to contrast the representational power of a point and a interval model of time. The conclusions from my comparison indicate that the interval time model is representationally more powerful in several ways:

1. Situations that cannot be described using the point model of time can be described using the interval model.
2. Certain reasoning tasks may be performed more efficiently using the interval model of time.

3. Certain tasks may be performed with the interval model of time that could not be performed using the point model.

Chapter 1 gave an overview of work that has been done in the area of representing time for digital design reasoning and stated the goals of this thesis. Chapter 2 presented the Helios Design Assistant as an example of system that uses a point time model for describing digital designs. Both the Helios design description language, Corona, and the simulator, MARS were described in detail. Chapter 3 defined the IRS predicates, presented the modifications to MARS that are needed to support IRS, and ended with a contrast of how one describes designs using the two time models.

Chapter 4 investigated, via case studies, different aspects of the interval time model's representational power. The first set of case studies demonstrated that the concept of *consecutiveness* can not be described using a point model of time. It also showed that to model tristate devices, *ad hoc* functionality had to be added to the point simulator, and that the added functionality served only to mimic properties of the interval time model. The second set of case studies demonstrated that a backward reasoning task was more efficient when modeled with intervals and that a class of assertions could be avoided when a point model of time was not used. The third section of chapter 4 demonstrated two tasks that could be performed using the interval model of time but not the point model. The first task was finding contention among different devices, and the second task was determining when no behavior was attributed to a device at a certain time. Chapter 4 ended with an argument that no representational power is lost when one switches from the point to the interval model of time.

Directions for Further Research

The implicit assumption of persistence, Hayes's frame problem, and the contrast between point and interval models of time are all closely related. It is conceivable that the power of intervals and the interval time model is derived from their ability to represent a frame or boundary around a span of time. One direction for further research would be an attempt to axiomatize the relative powers of the point and interval models of time, and show that the difference is exactly the ability to describe Hayes's frame axioms.

Ladkin is extending Allen's logic of convex sets of points, *i.e* intervals to sets of convex collections of points [Lad86a,Lad86b]. He is using his logic to describe events which recur regularly. An analysis of the representational power of his temporal model will need to be carried out, and the same questions as posed here will have to be asked about the relative powers of Allen's interval model and Ladkin's recurring model of time.

Bibliography

[AH84] J. Allen and P. Hayes. *A Common-Sense Theory of Time.* Technical Report, University of Rochester, 1984.

[All83] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), November 1983.

[Ama81] S. Amarel. On representations of problems of reasoning about actions. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 2 – 22, Morgan Kaufman, 1981.

[Fle80] W. I. Fletcher. *An Engineering Approach to Digital Design.* Prenctice-Hall Inc., 1980.

[Gen81] M. Genesereth. *The Use of Design Descriptions in Automated Dianosis.* Technical Report HPP-81-20, Stanford University, December 1981.

[Hay73] P. J. Hayes. The frame problem and related problems in artificial intelligence. In *Artificial and Human Thinking*, Jossey-Bass, San Francisco, 1973.

[Joy83] R. Joyce. *Reasoning about Time-dependent Behavior in a System for Diagnosing Digital Hardware Faults.* Technical Report HPP-83-37, Stanford University, 1983.

[Kra84] G. A. Kramer. *Brute Force and Complexity Management: Two Approaches to Digital Test Generation.* Master's thesis, MIT, 1984.

[Kra85] G. A. Kramer. Representing and reasoning about designs. In *IFIP Working Conference WG5.2*, Tokyo Japan, October 1985.

[Kra87] G. A. Kramer. Incorporating mathematical knowledge into design models. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 229 – 257, North-Holland, 1987.

[Lad86a] P. Ladkin. *Primitives and Units for Time specification.* Technical Report, Kestrel Institute, 1801 Page Mill Road Palo Alto CA 94394, March 1986.

[Lad86b] P. Ladkin. *Time Representation: A Taxonomy of Interval Relations.* Technical Report, Kestrel Institute, 1801 Page Mill Road Palo Alto CA 94394, March 1986.

[Mos83] B. C. Moszkowski. *Reasoning about Digitial Circuits.* PhD thesis, Stanford University, July 1983.

[Rus85] S. Russell. *The Compleat Guide to MRS.* Stanford University, 1985.

[Sin83] N. P. Singh. *MARS: A Multiple Abstraction Rule-Based, Simulator.* Technical Report 17, Schlumberger-CAS, 3340 Hillview Ave. Palo Alto CA 94304, December 1983.

[Sin84] N. P. Singh. *Corona: A Language for Describing Designs.* Technical Report, Schlumberger-CAS, 3340 Hillview Ave. Palo Alto CA 94304, September 1984.

[Sin85] N. P. Singh. *Exploiting Design Morphology to Manage Complexity.* PhD thesis, Stanford University, August 1985.

[Smi85] D. E. Smith. *Controlling Inference.* PhD thesis, Stanford University, August 1985.

[The81] *The TTL Data Book for Design Engineers.* Texas Instruments Incorporated, 2nd edition, 1981.