THE CRITTER SYSTEM: ANALYZING DIGITAL CIRCUITS BY PROPAGATING BEHAVIORS AND SPECIFICATIONS

Van E. Kelly
Louis I. Steinberg

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

ABSTRACT

CRITTER is a system that reasons about digital hardware designs, using a a declarative representation that can represent components and signals at arbitrary levels of abstraction. CRITTER can derive the behaviors of a component's outputs given the behaviors of the inputs, it can derive the specifications a component's inputs must meet in order for some given specifications on the outputs to be met, and it can verify that a given signal behavior satisfies a given specification. By combining these operations, it evaluates both the correctness and the robustness of the overall design.*

I INTRODUCTION: REASONING ABOUT DIGITAL CIRCUITS

In understanding or explaining a digital circuit, a human engineer's reasoning is flexible in several ways:

1. **Behavior vs. Specification:** The engineer can reason about what *will* happen in the circuit, given some input (its *behavior*), or about what *ought to* happen in order for the circuit to perform as desired (its *specifications*). In a correctly working circuit, the behavior must *satisfy* the specifications, but the two need not be identical; especially in matters of timing, "over-satisfaction" of some specifications is good practice.

2. **Forward vs. Backward:** The engineer can either use some fact about the *inputs* to a component to infer something about the *output* (forward reasoning), or vice-versa (backward reasoning).

3. **Level of Abstraction:** In reasoning, the engineer can treat each physical component separately or can view several components as a single functional module. Also, the engineer can view electrical signals at different levels of abstraction, e.g. as sequences of high/low voltages or as ASCII characters.

4. **Degree of Specificity:** The engineer can reason about the value of a signal either at a specific time or over all time (e.g., "The clock can *never* rise while this signal is high.") Also, the engineer can deal with specific data and time values, e.g. "10 nsec. after the clock rises", or general predicates, e.g. "between 5 and 10 nsec. after ...".

We have developed a system, called CRITTER, that displays some of this same flexibility. CRITTER does forward reasoning about behaviors and backward reasoning about specifications, propagating information step by step through a circuit in a manner reminiscent of constraint propagation systems [1, 2, 3, 4, 5, 6] and Waldinger's Goal Regression [7]. CRITTER handles varying levels of abstraction, statements about specific values or more general predicates, and statements quantified over all time.

CRITTER is a system for "critiquing" digital circuits. That is, it is designed to deductively solve the following problem:

> *Given the behavior of a circuit's inputs and the specifications on its outputs, determine the behavior and specifications for each signal in the circuit and whether each signal's behavior meets all its specifications.*

We also believe CRITTER's reasoning methods will prove valuable in other design tasks besides critiquing. In particular, our work on CRITTER has been part of a project on redesign aids [8]. We also envision applications in creating, debugging and documenting designs and in trouble-shooting physical hardware.

II THE ANATOMY OF CRITTER

In this section we will describe how CRITTER represents and reasons about circuits. First, however, we will introduce the circuit we will be using as a running example.

A. The Example Circuit

The circuit is shown in Fig. II-1 For purposes of exposition, it is a bit simpler than the real circuits on which we have used CRITTER. (See Sec. III)

This circuit is designed to convert ASCII alphanumeric characters to their EBCDIC equivalents. Every 576 nanoseconds a character appears at the input ASCII-IN, and is held there for 300 nsec.. After each character appears, the clock signal CLK will rise.

The actual translation to EBCDIC is done by ROM (read-only-memory) R, but this particular ROM requires its input to be available for at least 500 nsec. The input characters

*This material is based on work supported by the Defense Advanced Research Projects Agency under Research Contract N00014-81-K-0394. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

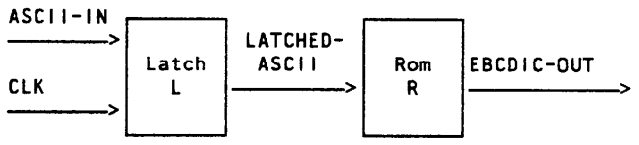

Figure II-1:
THE EXAMPLE CIRCUIT:
AN ASCII-EBCDIC CODE CONVERTER

are only available for 300 nsec. so we also have latch L which samples a character each time CLK rises and keeps it stable for the ROM until the next rise of CLK.

B. Representation of Circuits

The physical structure of a circuit is represented by *modules* and *data-paths*. A module represents either a single component or a cluster of components being viewed as a single functional block. In Fig. II-1, both the latch and the ROM are *modules*, and we could regard their composite as a module as well if we chose to do so. Similarly, a data-path represents either a wire or a group of wires.

1. Signals

The data flowing on a data path is viewed as a *data-stream*, which represents the entire history of the data on the path as an infinite sequence of data elements, i.e. as an array with subscripts running from 1 to infinity. Each element is characterized by a set of *features*, such as its TYPE, VALUE, START-TIME, and DURATION.

The *behavior* of a data-stream is represented by a formula for each feature, giving the value of the feature for an element as a function of the element's subscript. For instance, Fig. II-2 gives the set of behavior formulas for the ASCII characters at the input to the LATCH. (The syntax used by the CRITTER has been slightly modified here and in the other examples for the sake of readability.) Essentially Fig. II-2 says that

> For all i from 1 to infinity there exists a unique data element ASCII-IN(i), whose TYPE is ..., whose VALUE is ..., etc.

Note that all times in the circuit are given relative to some arbitrary T0.

```
NAME of signal      =  ASCII-IN
INDEX-STRUCTURE     =  ([i (from 1 to +INFINITY)])
TYPE(i)             =  ASCII-CHARACTER
VALUE(i)            =  NOT-SPECIFIED
START-TIME(i)       =  (576*i + 435) nsec. after T0
DURATION(i)         =   300 nsec.
```

Figure II-2:
GIVEN BEHAVIOR FOR ASCII-IN

A signal can also be described as an array of more than one dimension; the additional dimensions describe the substructure of the larger repeating elements. For example, Fig. II-3 gives the behavior for the other input signal, the clock pulse for the latch. It says that each element of the data-stream has subparts numbered 0 to 3, and gives formulas for the features of the subparts.

It is interesting to note that the two input signals are described at two very different levels of abstraction. The clock is described in terms of high and low voltages, which are even less abstracted than bits. The other input is described as a stream of characters, leaving implementation details implicit, including the fact that an ASCII character requires 7 bits to represent. and thus the LATCH module must really be 7 physical latches. The ability not only to abstract, but to mix levels of abstraction freely in a single circuit, is a powerful tool for focusing CRITTER's attention on the critical analytical issues within each circuit and suppressing insignificant detail.**

```
NAME of signal      =  CLK

INDEX-STRUCTURE     =  ([i (from 1 to +Infinity)]
                        [phase# (from  0 to 3)]  )

TYPE(i,phase#)      =  VOLTAGE-LEVEL

VALUE(i, phase#)    =  (CASE phase# of
                              0:   RISE
                              1:   HIGH
                              2:   FALL
                              3:   LOW)
START-TIME
   (i,phase#)       =  [576*(i+1) +
                          (CASE phase# of
                              0:   20
                              1:   21
                              2:   92
                              3:   93)]
                       nsec. after T0
DURATION
   (i,phase#)       =  (CASE phase# of
                              0:   1
                              1:   71
                              2:   1
                              3:   503)  nsec.
PERIOD              =  576 nsec.
```

Figure II-3:
GIVEN BEHAVIOR FOR CLK

2. Modules

A module's operation is described by a set of *operating conditions* and a set of *mappings*. An operating condition is a specification (i.e. a predicate) on a module's inputs that must be met if the module is to work. For example, in Figure II-4, operating condition [4] states that the clock for a given character must occur at least 20 nsec. after the character starts, i.e. the "setup time" for this latch is 20 nsec. Condition [6] says there must not be a clock between characters.

For each output of a module there is a mapping which gives a formula for each feature of the output in terms of

---

**Note, however, that CRITTER does not choose its own abstractions; they must be supplied from outside.

```
[1].  CLK must fit the definition of a
       STANDARD-CLOCK.
       (i.e. two dimensions, periodic, etc.)

[2].  INDEX-STRUCTURE of ASCII-IN must match
       '([* (from 1 to +Infinity)])'.

[3].  FOR ALL i > 0, j IN {1,3}
       DURATION[CLK (i j)] >=  20 nsec.

[4].  FOR ALL i > 0, the time of the
       NEXT (RISE of CLK) after
              START-TIME[ASCII-IN (i)]
      >= START-TIME[ASCII-IN (i)]
             + 20 nsec.

[5].  FOR ALL i > 0, the time of the
       NEXT (RISE of CLK) after
              START-TIME[ASCII-IN (i)]
      < START-TIME[ASCII-IN (i)]
         + DURATION[ASCII-IN (i)].

[6].  FOR ALL i > 0, the time of the
       NEXT (RISE of CLK) after
          (START-TIME[ASCII-IN (i)]
           + DURATION[ASCII-IN (i)])
      > START-TIME[ASCII-IN (i+1)].
```

Figure II-4:
OPERATING-CONDITIONS OF LATCH L

```
NAME of output     =  LATCHED-ASCII
INDEX-STRUCTURE    =  ([i (from 1 to +Infinity)])
TYPE(i)            =  TYPE[ASCII-IN(i)]
VALUE(i)           =  VALUE[ASCII-IN (i)]
START-TIME(i)      =  (the time of the
                        NEXT (RISE of CLK) after
                        START-TIME[ASCII-IN (i)])
                      + 35 nsec.
DURATION(i)        =  PERIOD[CLK] - 35 nsec.
```

Figure II-5:
INPUT/OUTPUT MAPPING OF LATCH L

features of the inputs. For example, Figure II-5 says that the output LATCHED-ASCII is a data-stream with elements from one to infinity, and tells how features of these elements depend on features of CLK and ASCII-IN. Note that a mapping is only reliable if all of the module's operating conditions are met by its inputs.

As another example, Fig. II-6 gives the operating conditions and mapping for ROM R.

C. Reasoning Methods

Using these representations, CRITTER can derive the behaviors of a module's outputs given the behaviors of the inputs, it can derive the specifications a module's inputs must meet in order for some given specifications on the outputs to be met, and it can verify that a given behavior satisfies a given specification. We will now discuss these kinds of reasoning.

OPERATING-CONDITIONS:

```
[1].  INDEX-STRUCTURE of input LATCHED-ASCII
      must match '([* (from 1 to +INFINITY)])'

[2].  FOR i > 0
      TYPE [LATCHED-ASCII(i)] =  ASCII-CHARACTER

[3].  FOR i > 0
       DURATION [LATCHED-ASCII (i)] > 500 nsec.
```

INPUT/OUTPUT MAPPING OF ROM:

```
NAME of output     =  EBCDIC-OUT
INDEX-STRUCTURE    =  ([i (from 1 to +Infinity)])
TYPE(i)            =  EBCDIC-CHARACTER
VALUE(i)           =  ASC-TO-EBCD
                        (VALUE [LATCHED-ASCII(i)])
START-TIME(i)      =  START-TIME [LATCHED-ASCII(i)]
                          + 500 nsec.
DURATION(i)        =  DURATION [LATCHED-ASCII (i)]
                          - 250 nsec.
```

Figure II-6:
DESCRIPTION OF OPERATION OF ROM R

1. Propagating Behaviors Forward

To calculate the behavior of a module's outputs, given the behavior of its inputs, one symbolically applies the module mappings to the inputs, by a process of substitution. The behavior of the output resembles the mapping, but with every reference to a feature of the input replaced by the corresponding formula from the input's behavior. For instance, to calculate the DURATION of the latched-characters we substitute the behavior of the CLK from Figure II-3:

```
PERIOD  = 576 nsec.
```

into the DURATION formula of Figure II-5:

```
DURATION(i)  = PERIOD(CLK) - 35 nsec.
             = 576 nsec - 35 nsec.
             = 541 nsec.
```

On the other hand, since nothing is specified in Figure II-2 about the VALUE of each incoming character, nothing can be substituted in the VALUE formula of the LATCH's I/O mapping, so the formula

```
VALUE(i)  = VALUE[ASCII-IN (i)]
```

is all that we know about that feature of the output behavior. A complete calculation of the behavior of the output of the latch is given in Figure II-7

Note that this substitution operation depends on the input *behavior* being represented as <feature> = <formula> rather than in the more general predicate form we use for operating conditions. The *mapping* must also be represented as <feature> = <formula>. If not, the substitution can still be done, but it will result in a behavior which does not have the <feature> = <formula> form and cannot be further propagated.

```
NAME of signal     =  LATCHED-ASCII
INDEX-STRUCTURE    =  ([i (from 1 to +INFINITY)])
TYPE(i)            =  ASCII-CHARACTER
VALUE(i)           =  VALUE[ASCII-IN (i)]
START-TIME(i)      =  [576*i + 631] nsec. after T0
DURATION(i)        =  541 nsec.
```

Figure II-7:
CALCULATED BEHAVIOR OF OUTPUT OF LATCH

Forward propagation can generate some messy expressions. CRITTER does some algebraic transformations to simplify expressions as much as it can. For instance, in calculating the START-TIME of LATCHED-ASCII it transforms

```
[NEXT (576*(j+1) + 20) AFTER (576*i + 435)] + 35
into
                    576*i + 631
```

Of course, the forward propagation is not valid unless the module's operating conditions are met, but this is checked as part of checking in general whether a data-stream meets its specifications, which is discussed below.

By repeated propagation, CRITTER can produce a behavior for each data-stream in a circuit, given behaviors for the circuit's inputs. Figure II-8 gives the calculated behavior of the ROM's output, produced by propagating the behavior of LATCHED-ASCII one step further.

```
NAME of signal     =  EBCDIC-OUT
INDEX-STRUCTURE    =  ([i (from 1 to +INFINITY)])
TYPE(i)            =  EBCDIC-CHARACTER
VALUE(i)           =  ASC-TO-EBCD
                        (VALUE [ASCII-IN (i)])
START-TIME(i)      =  (576*i + 1131) nsec. after T0
DURATION(i)        =  291 nsec.
```

Figure II-8:
CALCULATED BEHAVIOR OF ROM OUTPUT

2. Propagating Specifications Backward

Just as forward propagation computes the behaviors of the data-streams, CRITTER uses a process of *back propagation* to derive the specifications that these behaviors must meet. More specifically, given specifications (i.e. predicates) involving the output of any module, CRITTER can back propagate them, that is, derive a set of specifications involving the inputs to that module which are sufficient to ensure that the original output specification will be met. (This is a hardware analog of Dijkstra's "Weakest Precondition" [9]). Thus, given specifications involving the "global" outputs of a circuit, CRITTER can repeatedly back propagate them to produce specifications for all data-streams in the circuit.

Like forward propagation, back propagation is also done by a process of substitution. Each reference in the specification to a feature of the module's output is replaced by the corresponding formula from the module's mapping. For instance, Fig. II-9 gives a set of specifications for the output of our example circuit. Back-propagating specification [3] of Figure II-9 through the mapping of the ROM and simplifying gives:

```
For all j > 0,
   DURATION[EBCDIC-OUT(i)]              > 200      ==>
   DURATION[LATCHED-ASCII(i)] -  250 > 200      ==>
   DURATION[LATCHED-ASCII(i)]           > 450 nsec.
```

This substitution also depends on the *I/O mapping* being represented as <feature> = <formula>.

Of course, in order to produce the right output, the inputs to a module must first meet the operating conditions of the module. So, in addition to the specifications produced by substitution, the specifications on the input to a module must also include the operating conditions. It is this complete set of specifications which is further back propagated.

```
[1]  INDEX-STRUCTURE of EBCDIC-OUT must match
      '([i (from 1 to +INFINITY)])'

[2]  FOR ALL i > 0,
      VALUE(i) = ASC-TO-EBCD
                    (VALUE [ASCII-IN (i)])
[3]  FOR ALL i > 0,
      DURATION(i) > 200 nsec.

[4]  FOR ALL i > 0,
      START-TIME(i+1)
          - START-TIME(i) > 500 nsec.
```

Figure II-9:
SPECS INVOLVING STREAM EBCDIC-OUT

3. Checking Behaviors Against Specifications

Having derived the behavior and specifications for each data-stream, CRITTER must check to see if the specifications are met. In our case studies to date, CRITTER has been able to do this by replacing each feature-reference in the specification with the corresponding formula from the behavior, and then applying very straightforward simplifications to produce propositions which can be checked trivially. For instance, specification [4] in Figure II-9 reduces to the inequality 576 > 500, by simple symbolic subtraction of polynomials. It remains to be seen whether further examples will require more sophisticated proof techniques.

When the form of a specification is a single-sided arithmetic inequality, the margin by which that inequality is satisfied (e.g. 76 nsec. in specification [4]) represents a crude measurement of how *conservative* the circuit design is with respect to that specification. Thus CRITTER can determine not only if a design is correct (meets its specifications), but to some extent how robust it is.

Note that if we could do only forward propagation we could still verify the correctness of the circuit and get robustness measures for the circuit's outputs. If we could do only back propagation we could verify correctness and get robustness measures for the circuit's inputs. However, in order to get robustness estimates for an internal data-stream we need both its (forward progagated) behavior and all (back-propagated) specifications involving it.

III DISCUSSION

A. "In-Context" Module Descriptions

It should be noted that in our example we described our latch as if the only thing it could do is extend the duration of its input. However, a real latch can also be used, for instance, to select every other element from its input data-stream. What we have done is to place an artificial restriction on the input domain of our latch, in this case requiring there to be a clock pulse for each character. This then allowed us to use a much simpler mapping for the module. This is an example of a usage-specific or *in-context* module description. Such simplified mappings are much easier for people to write, and they result in less simplification being needed during propagation, but what we wind up with is not really a representation of a "latch" but rather of a "latch used as duration extender".

As long as the behaviors of all input signal of the module obey the more restrictive "in-context" operating conditions, the result of forward propagation is unchanged by the simplification. However, back propagation of specifications will produce input pre-conditions that are not as weak as possible. In verifying a design that is already complete and correct, this is not necessarily bad; it just means that CRITTER may be overly conservative in estimating robustness. When using CRITTER incrementally, however, to evaluate fragments of a design in progress, over-strong pre-conditions might entail frivolous and arbitrary restrictions on possible methods for completing the design. It remains to be seen how much of a problem this is in practice.

It is uncertain if the set of all common component "usages" is small enough that a library of them would be useful, or if in-context descriptions should be generated individually only as needed. We are currently working on methods to automatically generate in-context descriptions from out-of-context descriptions.

B. Data Abstraction

As we have seen, data-streams can be described at various levels of abstraction. In fact, the module that produces a data-stream may employ a different abstraction from the module that uses it. For instance, a counter might be described as transmitting a sequence of integers while a latch which receives these numbers might consider them simply as vectors of bits. We handle this kind of translation by interposing a type-converter *pseudo-module* to convert the numbers to bit vectors. A pseudo-module is represented just like any other module, but it corresponds to a change in interpretation, rather than to a physical component. The same propagation methods work on pseudo-modules as on real modules.

C. Limitations

1. Feedback and State

CRITTER still has a number of limitations. One major problem is that it cannot handle circuits with inter-module feedback. That is, it can only handle circuits in which the relation "module A uses the output of module B" induces a partial order. This is a less drastic limitation than it might appear since there are many useful and interesting circuits that have no inter-module feedback (i.e. where feedback loops exist, they are enclosed within a single module).

None the less, this is a serious limitation, and we place a high priority on removing it. We plan to look at both the previous work on cycles in constraint networks (e.g. [1, 3, 4, 5, 6]) and the work on using loop invariants in program verification [9, 10].

Another limitation is that we do not deal explicitly with the internal state of a component such as a RAM or a latch. This, like internal feedback loops, can be concealed from CRITTER by choosing a suitably abstract "in-context" description for a module. We expect that once we can handle feedback, we can model internal state as an implicit feedback path, as is done with formal Finite State Automata.

2. Uncertainty

Our "<feature>=<formula>" notation for behaviors and mappings implies that we know a precise formula for each feature. However, for parameters like delay through a component, the parts catalogue normally gives a range of possible values. One solution to this is to embed *uncertainty* in these formulas by introducing *slack variables* to represent it, e.g.

```
for all i > 0,
        START-TIME (out (i)) =
              (START-TIME (in (i))) + SLACK1
```

The restrictions on the uncertainty, e.g. 10 nsec < $SLACK_1$ < 20 nsec, can be carried along separately.

D. Relation to Constraint Propagation

Our desire to do propagation both forward and backward and to eventually handle feedback lead us to view CRITTER as a kind of constraint propagation system. However, much previous constraint-system work [1, 3, 4, 5, 6] has focused on problems of finding consistent solutions to complex networks of constraints, and has assumed that the individual values and constraints were quite simple. In contrast, our focus has been more like that of Stefik [11], in that we have put off dealing with complex networks (e.g. feedback), but have dealt with richer kinds of values and constraints. In particular:

- Because we need to deal with entire time histories of signals and even arbitrary predicates on these time histories, rather than with single values or small sets of values, our individual propagation steps involve symbol substitution rather than, e.g., arithmetic operators. In fact, we really have two kinds of propagation, one for behaviors and another for specifications, but we can use the same representation of our constraint (i.e. the module function) and just apply different processes to it for the two kinds of propagation.

- Since a specification really expresses a constraint among several data-streams (e.g. conditions 4-6 in Fig. II-4), back-propagation is really the propagation of one kind of constraint (a specification) through another kind of constraint (a function). We could even express both these constraints in the same language, except that in order to do the substitution, the function mapping has to be in a <feature> = <formula> form.

E. Early Experiences With CRITTER

CRITTER has been used to test a fragment of a mature circuit design (1976) for a TTL-based CRT terminal video controller. About thirty specifications in all had to be satisfied, on a total of nine DATA-STREAMS. Surprisingly, it quickly discovered a potential timing anomaly that had never been noticed before in conventional testing, nor in actual use of the circuit.

IV CONCLUSIONS

CRITTER thus embodies varied and useful kinds of reasoning about digital circuits. These reasoning abilities are useful for automatically critiquing a circuit, and should be applicable to several other tasks as well. One pressing need is to extend CRITTER to handle circuits with feedback and components with state. We also plan to implement mechanisms to handle the slack variables, and to try to apply CRITTER's reasoning methods to other tasks, such as trouble-shooting and automatic design.

V Acknowledgments

The work reported here was done as part of the Digital Design Project at Rutgers University. Other members of the group, including Tom Mitchell, Pat Schooley, Jeff Shulman, and Tim Weinrich, have made significant contributions to the ideas and programs discussed above. Tom Mitchell and Pat Schooley also made a number of particularly helpful comments on earlier drafts of this paper.

References

[1] Borning, A. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory." *ACM Transactions on Programming Languages and Systems.* 3:4 October (1981) 353-387.

[2] de Kleer, Johan and Gerald J. Sussman "Propagation of Constraints Applied to Circuit Synthesis", Memo No. 485, M.I.T., September 1978.

[3] de Kleer, Johan, *Causal and Teleological Reasoning in Circuit Recognition,* PhD dissertation, M.I.T., January 1979.

[4] Stallman, R.M. and Sussman, G.J. "Forward Reasoning and Dependency-Directed Backtracking in a System for Conputer-Aided Circuit Analysis." *Artificial Intelligence.* 9:2 October (1977) 135-196.

[5] Steele, G.L., *The Definition and Implementation of a Computer Programming Language Based on Constraints,* PhD dissertation, M.I.T., August 1980.

[6] Steels, L. "Constraints as Consultants", AI Memo 14, Schlumberger-Doll Research, December 1981.

[7] Waldinger, R.J., "Achieving Several Goals Simultaneously," in *Machine Intelligence 8,* Elcock, E. and Michie, D., ed., Ellis Horwood, Chichester, 1977, 94-136, MI8

[8] Mitchell, T. , Steinberg, L., Smith, R.G., Schooley, P., Kelly, V. and Jacobs, H. "Representations for Reasoning About Digital Circuits." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* 1 (1981) 343-344, IJCAI7

[9] Dijkstra, Edsger W. *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1976.

[10] Greif, Irene and Meyer, Albert R. "Specifying the Semantics of while Programs: a Tutorial and Critique of a Paper by Hoare and Lauer." *ACM Transactions on Programming Languages and Systems.* 3:4 October (1981) 484-507.

[11] Stefik, M.J., *Planning With Constraints,* PhD dissertation, Stanford University, January 1980, CSD REPORT Stan-CS-80-784