An Introduction to CTL

©Kathi Fisler, 2001
CS525V

January 21, 2001

1 Introduction and Lecture Goals

In formal verification, we want to prove that designs satisfy certain requirements. This requires that we agree on a rigorous notation for expressing both designs and their requirements (aka properties). This lecture introduces you to the models we will use for designs and the logic (language) we will use for properties.

2 A Simple Problem

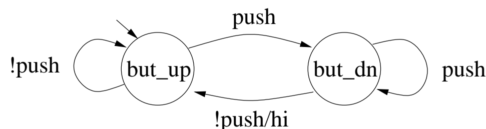
Consider a simple device whose interface consists of a single button. When someone releases the button, the device beeps (or alternatively, says "hi"). What's important about this device is that it only says "hi" once for each push of the button. Such devices are useful in real systems; in an elevator, for example, even if someone holds the button down for several seconds, we only want to summon the elevator once.

The desired requirements on our device are as follows:

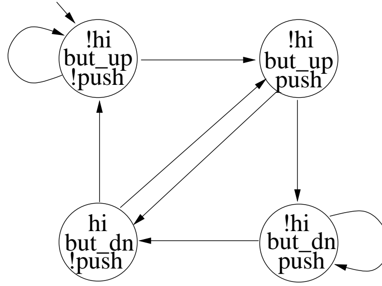
- If someone releases the button, the device says hi
- The hi only happens for one unit of time
- The device produces at most one high per button push

2.1 Modeling the Device

We can model the device using a simple state machine. The labels on the edges indicate the inputs that enable the transitions. The edge marking "!push/hi" says "if push is false, take this transition and output hi".

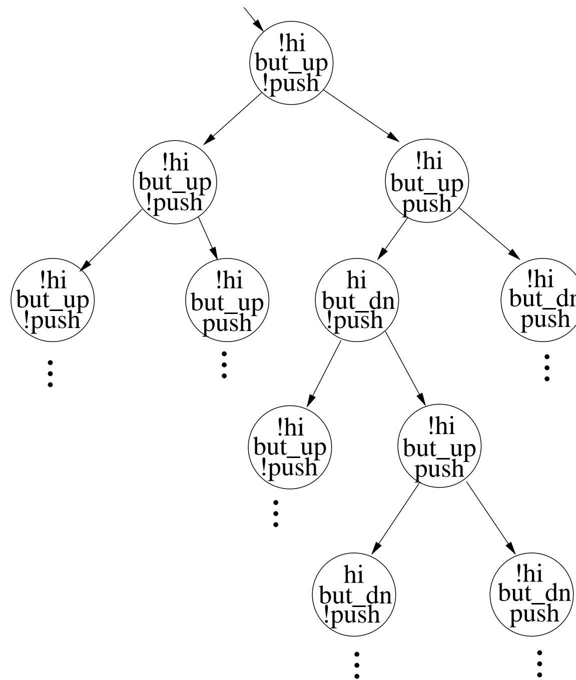


This kind of state machine, where outputs are associated with edges, is called a *Mealy machine*. For purposes of motivating verification and our requirements logic, it will help to write this machine such that all of the inputs and outputs are associated with states rather than with transitions. Here is the equivalent state machine with state-associated variables. In this version, all edges are implicitly labeled "true". This machine is non-deterministic while the original machine was deterministic.



2.2 Stating the Requirements Formally

To start thinking about a more formal language for expressing the requirements for our simple device, let's unroll the previous state machine into a tree showing all of the possible paths through a run of this device. Note that the tree is necessarily infinite:



2.2.1 Stating Requirements in Terms of the Tree

Let's return to our stated requirements and try to rephrase them as statements about the tree:

- If someone releases the button, the device says hi

  "At all states labeled but_dn and !push, hi is also true"

  The phrase "states labeled but_dn and !push" captures the button release. Since the original requirement doesn't require the "hi" to happen at exactly the same time as the release, we can generalize this slightly to

  "At all states labeled but_dn and !push, hi is eventually also true"

- The hi only happens for one unit of time

  "In all states, if hi is true, then hi is false in the next state"

- The device produces at most one high per button push

  "If hi is true, it isn't true again until the button has been pushed"

2.2.2 Moving the Requirements Towards Propositional Logic

Our goal is to uncover a formal logic for expressing requirements. The tree-based statements already suggest using the standard operators from propositional logic. Let's rewrite the tree-based versions with those operators first:

- At all states labeled but_dn and !push, hi is eventually also true

  At all states, but_dn $\wedge$!push $\rightarrow$ eventually hi

- In all states, if hi is true, then hi is false in the next state

  In all states, hi $\rightarrow$!hi in the next state

- If hi is true, it isn't true again until the button has been pushed

  hi $\rightarrow$!hi until push

Now, what of our requirements is not expressed in formal logic? It's phrases about states ("in all states") and about time ("eventually", "next", "until"). Thus, in order to turn these requirements fully into logic, we need to augment our logic with notions of time.

2.2.3 Moving the Requirements Towards Temporal Logic

We will use the following symbols to stand for different time phrases:

"globally"	**G**
"eventually"	**F**
"next"	**X**
"until"	**U**

Let's take another pass over our requirements, using these new operators in place of the temporal phrases:

- At all states, but_dn $\wedge$!push $\rightarrow$ eventually hi

  G (but_dn $\wedge$!push $\rightarrow$ **F** hi)

- In all states, hi $\rightarrow$!hi in the next state

  G (hi $\rightarrow$ **X** !hi)

- hi $\rightarrow$!hi until push

  hi $\rightarrow$ [!hi **U** push]

These statements now look fully formal – all parts of the expressions are either operators in our logic or variables in our design. There's one small problem remaining though. Go back to the tree. Consider the statement "**G** (hi $\rightarrow$ **X** !hi)". If I'm in a state where hi is true, do I need to look for !hi in *some* next state or in *all* next states? The formal requirement isn't clear, but that should be an important distinction. Therefore, we need to augment our formal requirements slightly so that for every temporal operator, they indicate whether the formula should be true on all paths starting from a state, or only on some of them. We will use **A** for "all paths" and **E** for "some path".

Updating our requirements with path operators, we now get the following formal requirements:

- At all states, but_dn $\wedge$!push $\rightarrow$ eventually hi

  AG (but_dn $\wedge$!push $\rightarrow$ **AF** hi)

- In all states, hi $\rightarrow$!hi in the next state

  AG (hi $\rightarrow$ **AX** !hi)

- hi $\rightarrow$!hi until push

  hi $\rightarrow$ A[!hi **U** push]

  Notice that for until, the path quantifier goes outside the square brackets, not with the **U**.

This logic, including the propositional and temporal operators and path quantifiers is called CTL (for "computation tree logic").

2.2.4 One Last Set of Changes to the Requirements

The current statements of the requirements are fully expressed in temporal logic. Here are two small critiques related to writing good and useful properties though.

1. Ideally, requirements should be phrased in terms of the interface variables, not the internal states of the design. This makes requirements reusable across several designs for the same device. Our first requirement:

$$\mathbf{AG}\ (\text{but_dn} \wedge \text{!push} \rightarrow \mathbf{AF}\ \text{hi})$$

   refers to but_dn. How can we phrase this only in terms of push and hi? We used $\text{but_dn} \wedge \text{!push}$ to capture a button release. We can also capture that by seeing a push in one state and !push in the next. We therefore rewrite the first requirement as follows:

$$\mathbf{AG}\ (\text{push} \rightarrow \mathbf{AX}\ (\text{!push} \rightarrow \mathbf{AF}\ \text{hi}))$$

2. Our third requirement, "$\text{hi} \rightarrow \text{!hi}$ until push" doesn't say that it needs to be true in all states. As written, this says that the requirement only has to hold once. It's rare that you'll have a property that only has to happen once; expect your properties to always start with some temporal operator (and its path quantifier). We'll edit the property to require it to hold in all states:

$$\mathbf{AG}(\text{hi} \rightarrow \text{A}[\text{!hi}\ \mathbf{U}\ \text{push}])$$

2.2.5 The Final Requirements

Here's the final set of requirement statements for our simple device:

- $\mathbf{AG}\ (\text{push} \rightarrow \mathbf{AX}\ (\text{!push} \rightarrow \mathbf{AF}\ \text{hi}))$
- $\mathbf{AG}\ (\text{hi} \rightarrow \mathbf{AX}\ \text{!hi})$
- $\mathbf{AG}(\text{hi} \rightarrow \text{A}[\text{!hi}\ \mathbf{U}\ \text{push}])$

3 Where Do We Go From Here?

This lecture gives you just the motivational flavor for CTL and some examples of its syntax. We still need to understand the semantics of CTL (when a CTL formula is true of a tree) and the algorithm that checks whether a CTL formula is true of a state machine.