22c181:
Formal Methods in Software Engineering

The University of Iowa
Spring 2008

Verifying Safety Property of Lustre Programs: Temporal Induction

Semantics of Lustre programs

- A Lustre program is in essence a set of constraints between its input streams and output streams.
- These constraints operate in an algebra of streams
- But they can also seen as Boolean and arithmetic constraints over instantaneous configurations of the program.

Important observation: A stream x containing values of type T is essentially a function $x : \mathbb{N} \rightarrow T$.
For each $n \in \mathbb{N}$,

$$x(n)$$

is the value of x at position (or, *instant*) n.

Instantaneous Configuration

Let L be a Lustre program. Let

- $x_1, \ldots, x_p$ be streams given in input to L, and
- $x_{p+1}, \ldots, x_{p+q}$ be the non-input (i.e., local and output) streams computed by P.

For each $n \in \mathbb{N}$, the tuple of values

$$\langle x_1(n), x_2(n), \ldots, x_{p+q}(n) \rangle$$

is a *configuration (of L at instant n)*.

Instantaneous Configuration: Example

```
node counter (R:bool, X:int) returns (Y:bool);
var C: int;
let
  C = X -> if R then X else pre(C) + 1;
  Y = (C = 5);
tel;
```

			0	1	2	3	4	5	...
R	$=$	(	false,	false,	false,	true,	false,	false,	...
X	$=$	(	0,	4,	5,	1,	0,	11,	...
C	$=$	(	0,	1,	2,	1,	2,	3,	...
Y	$=$	(	false,	false,	false,	false,	false,	false,	...

$\langle R(3), X(3), C(3), Y(3)\rangle = \langle \mathit{true}, 1, 1, \mathit{false}\rangle$ is the configuration at instant 3.

Instantaneous Configuration: Example

```
node counter (R:bool, X:int) returns (Y:bool);
var C: int;
let
  C = X -> if R then X else pre(C) + 1;
  Y = (C = 5);
tel;
```

The program above can be seen as the following set of constraints for all $n \in N$.

$$
\begin{aligned}
C(n) &= \text{if } n = 0 \text{ then } X(n) \\
&\quad\ \text{else if } R(n) \text{ then } X(n) \\
&\quad\ \text{else } C(n-1) + 1 \\
Y(n) &= C(n) = 5
\end{aligned}
$$

Proving Properties

```
node test( X: bool ) returns ( P : bool );
var A, B : bool;
let
  A  = X -> pre A;
  B  = not (not X -> pre (not B));

  --- A and B are identical streams
  P = A = B;
tel;
```

Conjecture: `test` always returns that constantly true stream.

How do we prove that?

Proving Properties by Induction

- Mathematically, the program `test` expresses, for all $n \in \mathbb{N}$, the constraints set Δ_n:

$$
\begin{array}{rcl}
A(n) & = & \text{if } n = 0 \text{ then } X(n) \text{ else } A(n-1) \\
B(n) & = & \text{not (if } n = 0 \text{ then not } X(n) \text{ else not } B(n-1)) \\
P(n) & = & A(n) = B(n)
\end{array}
$$

- We want to show that $P(n) = true$ for all $n \in \mathbb{N}$.
- To do that, we can reason by induction on n:
  1. First, we prove that $P(0)$'s value is always $true$.
  2. Then, we prove that whenever $P(n)$ is $true$ for an arbitrary n then $P(n+1)$ is also $true$.

Proving Properties by Induction

Induction proof:

Base case) Prove that $\Delta_0 \Rightarrow P(n)$

Induction Step) Prove that $\Delta_n \wedge \Delta_{n+1} \wedge P(n) \Rightarrow P(n+1)$

We have 3 possibilities.

1. Both the base case and the induction step hold. Then, we can conclude that P is always $true$.
2. The base case does not hold. Then, clearly, P is sometimes $false$.
3. The base case holds but the induction step does not. Then, we cannot conclude anything about P.

Induction Proof: Example

$$\Delta_n: \quad \begin{array}{rcl} A(n) & = & \text{if } n = 0 \text{ then } X(n) \text{ else } A(n-1) \\ B(n) & = & \text{not (if } n = 0 \text{ then not } X(n) \text{ else not } B(n-1)) \\ P(n) & = & A(n) = B(n) \end{array}$$

Base case) Δ_0 is equivalent to:

$$\begin{array}{rcl} A(0) & = & X(0) \\ B(0) & = & \text{not (not } X(0)) \\ P(0) & = & A(0) = B(0) \end{array}$$

Clearly, $P(0) = true$

Induction Proof: Example

$$
\Delta_n: \quad
\begin{aligned}
A(n) &= \text{if } n = 0 \text{ then } X(n) \text{ else } A(n-1) \\
B(n) &= \text{not (if } n = 0 \text{ then not } X(n) \text{ else not } B(n-1)) \\
P(n) &= A(n) = B(n)
\end{aligned}
$$

Induction Step) Assume that $A(n), B(n), C(n)$ are defined as in Δ_n. Δ_{n+1} is equivalent to:

$$
\begin{aligned}
A(n+1) &= A(n) \\
B(n+1) &= \text{not (not } B(n)) \\
P(n+1) &= A(n+1) = B(n+1)
\end{aligned}
$$

If we assume that $P(n)$ is $true$, it must be that $A(n) = B(n)$.

But then, we can conclude that $P(n+1)$ is $true$.

Limits of Simple Induction

```
node counter (R: bool) returns (P: bool);
var C: int;
let
  C =  0 -> if (R or pre(C) = 2) then 0
            else pre(C) + 1;

  P =  C <= 4;
tel;
```

Observe:

- `C` is never more than 2, so `P` is constantly `true`.
- However, simple induction is unable to prove that.
- The problem is that the induction step does not hold.

Why the induction step does not hold

$$\Delta_n: \quad \begin{array}{rcl} C(n) & = & \text{if } n = 0 \text{ then } 0 \text{ else} \\ & & \text{if } R(n) \text{ or } C(n-1) = 2 \text{ then } 0 \text{ else } C(n-1) + 1 \\ P(n) & = & C(n) \leq 4 \end{array}$$

$$\Delta_{n+1}: \quad \begin{array}{rcl} C(n+1) & = & \text{if } R(n+1) \text{ or } C(n) = 2 \text{ then } 0 \text{ else } C(n) + 1 \\ P(n+1) & = & C(n+1) \leq 4 \end{array}$$

We need to show that the following implication holds:

$$\Delta_n \wedge \Delta_{n+1} \wedge P(n) \;\Rightarrow\; P(n+1) \quad (*)$$

However, if we set, e.g., n to 10, $C(n-1)$ to 3, and $R(n)$ and $R(n+1)$ to $false$, we can satisfy $\Delta_n \wedge \Delta_{n+1} \wedge P(n)$ and falsify $P(n+1)$.

Why the induction step does not hold

$$\Delta_n: \quad \begin{array}{rcl} C(n) & = & \text{if } n = 0 \text{ then } 0 \text{ else} \\ & & \text{if } R(n) \text{ or } C(n-1) = 2 \text{ then } 0 \text{ else } C(n-1)+1 \\ P(n) & = & C(n) \leq 4 \end{array}$$

$$\Delta_{n+1}: \quad \begin{array}{rcl} C(n+1) & = & \text{if } R(n+1) \text{ or } C(n) = 2 \text{ then } 0 \text{ else } C(n)+1 \\ P(n+1) & = & C(n+1) \leq 4 \end{array}$$

Problem:

- a value of 3 for $C(n-1)$ is impossible in the program
- but the premise of $(*)$ is not strong enough to rule it out

Why the induction step does not hold

$$\Delta_n: \begin{array}{rcl} C(n) & = & \text{if } n = 0 \text{ then } 0 \text{ else} \\ & & \text{if } R(n) \text{ or } C(n-1) = 2 \text{ then } 0 \text{ else } C(n-1)+1 \\ P(n) & = & C(n) \leq 4 \end{array}$$

$$\Delta_{n+1}: \begin{array}{rcl} C(n+1) & = & \text{if } R(n+1) \text{ or } C(n) = 2 \text{ then } 0 \text{ else } C(n)+1 \\ P(n+1) & = & C(n+1) \leq 4 \end{array}$$

Problem:

- a value of 3 for $C(n-1)$ is impossible in the program
- but the premise of $(*)$ is not strong enough to rule it out

Solution:

- look at a few more preceding configurations

k*-induction: Induction with Depth*

Fix some $k \geq 0$

Base case) Prove that

$$\Delta_0 \wedge \cdots \wedge \Delta_k \;\Rightarrow\; P(0) \wedge \cdots \wedge P(k)$$

Induction Step) Prove that

$$\Delta_n \wedge \cdots \wedge \Delta_{n+k+1} \wedge P(n) \wedge \cdots \wedge P(n+k) \;\Rightarrow\; P(n+k+1)$$

We have again 3 possibilities:

k-induction: Induction with Depth

Fix some $k \geq 0$

Base case) Prove that

$$\Delta_0 \land \cdots \land \Delta_k \;\Rightarrow\; P(0) \land \cdots \land P(k)$$

Induction Step) Prove that

$$\Delta_n \land \cdots \land \Delta_{n+k+1} \land P(n) \land \cdots \land P(n+k) \;\Rightarrow\; P(n+k+1)$$

We have again 3 possibilities:

1. Both the base case and the induction step hold. Then, we can conclude that P is always $true$.

k-induction: Induction with Depth

Fix some $k \geq 0$

Base case) Prove that

$$\Delta_0 \wedge \cdots \wedge \Delta_k \;\Rightarrow\; P(0) \wedge \cdots \wedge P(k)$$

Induction Step) Prove that

$$\Delta_n \wedge \cdots \wedge \Delta_{n+k+1} \wedge P(n) \wedge \cdots \wedge P(n+k) \;\Rightarrow\; P(n+k+1)$$

We have again 3 possibilities:

2. The base case does not hold.
   Then, P is $false$ for some $m \in \{0, \ldots, k\}$.

k-induction: Induction with Depth

Fix some $k \geq 0$

Base case) Prove that

$$\Delta_0 \wedge \cdots \wedge \Delta_k \;\Rightarrow\; P(0) \wedge \cdots \wedge P(k)$$

Induction Step) Prove that

$$\Delta_n \wedge \cdots \wedge \Delta_{n+k+1} \wedge P(n) \wedge \cdots \wedge P(n+k) \;\Rightarrow\; P(n+k+1)$$

We have again 3 possibilities:

3. The base case holds but the induction step does not. Then, we cannot conclude anything about P.

   But we can increase k and start again.

Previous Example

$$\Delta_n: \quad \begin{aligned} C(n) &= \text{if } n = 0 \text{ then } 0 \text{ else} \\ &\quad \text{if } R(n) \text{ or } C(n-1) = 2 \text{ then } 0 \text{ else } C(n-1)+1 \\ P(n) &= C(n) \leq 4 \end{aligned}$$

$$\Delta_{n+1}: \quad \begin{aligned} C(n+1) &= \text{if } R(n+1) \text{ or } C(n) = 2 \text{ then } 0 \text{ else } C(n)+1 \\ P(n+1) &= C(n+1) \leq 4 \end{aligned}$$

- With k-induction we can prove that P is always $true$.
- **Exercise:** Find the smallest value of k that will do.

The k-induction Procedure

```
k := 0;
while true do
  check validity of
  Δ_0 ∧ ··· ∧ Δ_k ⇒ P(0) ∧ ··· ∧ P(k);
  if counter-example found then
    return counter-example
  end if;
  check validity of
  Δ_n ∧ ··· ∧ Δ_{n+k+1} ∧ P(n) ∧ ··· ∧ P(n+k) ⇒ P(n+k+1);
  if valid then
    return "Property holds"
  end if;
  k := k + 1;
end while
```

Features of the k-induction Procedure

- When Δ contains no multiplications, the validity tests in lines 3 and 7 can be performed completely automatically.
- The induction procedure is *sound*: if it says that the property holds, then the property does hold.
- However, the procedure is still *incomplete*: for some properties that do hold it may loop forever.
- The procedure can be made complete for some (large) classes of Lustre programs, including *finite state* ones.
- However, it is impossible to make the procedure complete (and still automatic) for all Lustre programs.

Limits of k-induction

```
node counter2 (R, X: bool) returns (P: bool);
var C: int; let
  C =  0 -> if (R or pre(C) = 2) then 0
            else pre(C) + 1;
  P =  X or (C <= 4);
tel;
```

Observe:

- Similar to `counter` but now `P` is `X or (C <= 4)` instead of `C <= 4`, with `X` an additional input stream.
- `P` is always `true` but k-induction is unable to prove that for any k.
- For each k, there is a counter-example for the induction step, e.g., $n = 10$, $C(n-1) = 4$, $X(n) = true$, ..., $X(n+k) = true$, and $X(n+k+1) = false$.

A Simplifying Assumption

Let us consider only Lustre programs where `pre` applies only to variables.

Note: This is with no loss of generality. For example, the first program below can be rewritten equivalently into the second:

```
node Foo (X,Y: int) returns (Z:int);
let
  Z = 0 -> pre (X + Y);
tel;

node FooNorm (X,Y: int) returns (Z:int);
var U: int
let
  U = X + Y;
  Z = 0 -> pre(U);
tel;
```

Program State

If L is a Lustre program, let $\mathbf{S}$ be the tuple of L's *state variables*, non-input variables that occur within a `pre`.

Example. $\mathbf{S} = \langle A, C \rangle$ for this program:

```
node test( X: bool ) returns ( P : bool );
var A, B, C : bool;
let
  A  = X -> pre A;
  B  = not (not X -> pre(C));
  C  = not B;
  P  = A = B;
tel;
```

The value $\mathbf{S}_n$ that the tuple $\mathbf{S}$ has at some instant n is the *state of L at instant n*.

k-induction with Distinct States

- We can make k-induction *less incomplete*, by considering only configurations with distinct states.
- Let $D_{0,k}$ be the formula stating that the states $\mathbf{S}_0, \ldots, \mathbf{S}_k$ are pairwise distinct. (And similarly for $D_{n,n+k+1}$).
- We can use

  Base case)

$$D_{0,k} \wedge \Delta_0 \wedge \cdots \wedge \Delta_k \Rightarrow P(0) \wedge \cdots \wedge P(k)$$

  Induction step)

$$D_{n,n+k+1} \wedge \Delta_n \wedge \cdots \wedge \Delta_{n+k+1} \wedge P(n) \wedge \cdots \wedge P(n+k) \Rightarrow P(n+k+1)$$

The k-induction Procedure with Distinct States

$k := 0$;
while *true* **do**
  `check validity of` $D_{0,k} \wedge \Delta_0 \wedge \cdots \wedge \Delta_k \Rightarrow P(0) \wedge \cdots \wedge P(k)$;
  if counter-example found **then**
    return `counter-example`
  end if;
  `check validity of`
  $D_{n,n+k+1} \wedge \Delta_n \wedge \cdots \wedge \Delta_{n+k+1} \wedge P(n) \wedge \cdots \wedge P(n+k) \Rightarrow P(n+k+1)$;
  if valid **then**
    return `"Property holds"`
  end if;
  $k := k + 1$;
  `check validity of` $\Delta_0 \wedge \cdots \wedge \Delta_k \Rightarrow \neg D_{0,k}$;
  if valid **then**
    return `"Property holds"`
  end if;
end while

k-induction with Distinct States

- Adding the distinct states restriction to k-induction preserves its soundness.
- It makes it complete for programs where every legal execution sequence with pairwise distinct states is shorter than some positive integer d.
- This is the case, for instance, for *finite state programs*, programs whose state variables can take only finitely many values.
- But it is also the case for some infinite state programs like `counter2`.