Programming Embedded Systems

Lecture 11

Lustre V&V

Monday Feb 20, 2012

Philipp Rümmer
Uppsala University
`Philipp.Ruemmer@it.uu.se`

Lecture outline

- Formalisation of requirements in Lustre
- Synchronous observers
- Static V&V of Lustre programs
  (using Luke)

Recap: Lustre

- Synchronous dataflow language, textual
- Basic building block: nodes consisting of flow definitions
- Basic datatypes: `bool`, `int`
- Example: integer register

```
node IntRegister(newValue : int; store : bool)
     returns (val : int);
```

Recap: correctness

- Software is called **correct** if it complies with its specification
  - Often: spec. is a set of requirements and/or use cases
- Software that violates spec. contains **bugs/defects**
- Correctness of software can be verified

Recall

- What are
  - **static** and
  - **dynamic** analysis methods?

Mathematical techniques like proving, model checking (used in this lecture)

Mostly: testing, simulation

Typical V&V in Lustre

- **Safety requirements** are first formulated as text (in, say, English)
- Textual requirements are translated to Lustre expressions
- Formal requirements are attached to Lustre program in form of **synchronous observers**
- Correctness of Lustre program is checked using **testing** or **model checking**

Corresponds to “Safety module” in Elevator lab

Synchronous observers

- A synchronous observer for a node

  node Prog(parameters) **returns**(vals);

  is a Lustre node of the shape

```
node ReqProg(parameters)
     returns(ok1, ok2, … : bool);
var vals;
let
   (vals) = Prog(parameters);
   ok1 = requirement1;
   ok2 = requirement2;
   ...
tel
```

Formalised requirements, talking about `parameters` and `vals`

Example: multi-state switch

```
node MultiStateSwitch(pin0 : bool) returns (pin1, pin2 : bool);
var n : int;
let
  n = ResetCounter(true, not pin0);
  pin1 = n > 1 and n < 20;
  pin2 = n >= 20;
tel
```

- Example requirements:
  - `pin1` and `pin2` are never true at the same time
  - `pin1` and `pin2` are true only if `pin0` is true

Verification using Luke

- **Simulation**:

  ```
  luke --node top_node filename
  ```

- **Verification**:

  ```
  luke --node top_node --verify filename
  ```

  - returns either

    “`Valid. All checks succeeded. Maximal depth was n`”

    or

    “`Falsified output ‘X’ in node ‘Y' at depth n`”

    along with a counterexample.

What does “All checks succeeded” mean?

- Intuitively:
  A mathematical proof has been found that the synchronous observer **never** returns false
- Implies:
  Requirements **cannot** be violated

What does “All checks succeeded” mean? (2)

- Different from testing:
  - **All** possible program inputs have been considered
  - **However**: only meaningful under assumption that compiler + hardware is correct
    → **realistic?**
- Luke uses SAT-based model checking + k-induction *(more details later)*

Counterexamples

- Give diagnostic feedback if requirements can be violated
- Example in MultiStateSwitch:
    - `pin2` is never true

Does not actually hold

Formalisation of requirements

From text to Lustre expressions

- Textual requirements often use patterns with commonly understood meaning
- But: text is not always unambiguous; writing good/precise requirements can be difficult
- (Similarly:
  Text to C expressions, Elevator lab)

Common English patterns

English	Logic	Lustre (similar for C)
A and B A but B	A & B	A and B
A if B A when B A whenever B	...	...
if A, then B A implies B A forces B		
only if A, B B only if A		
A precisely when B A if and only if B		
A or B either A or B		
A or B		

Ambiguous; to clarify, write "either A or B" or "A or B, or both"

Common English patterns

English	Logic	Lustre (similar for C)
A and B A but B	A & B	A and B
A if B A when B A whenever B	B => A	B => A
if A, then B A implies B A forces B	A => B	A => B
only if A, B B only if A	B => A	B => A
A precisely when B A if and only if B	A <=> B	A = B
A or B either A or B	A (+) B (exclusive or)	A xor B
A or B	A v B (logical or)	A or B

Ambiguous;
to clarify, write
"either A or B"
or
"A or B, or both"

Temporal requirements

- Patterns on previous slides are on the **propositional** level
- Requirements often contain **temporal** statements
- Example in MultiStateSwitch:
  - if `pin2` is true, then `pin1` has been true sometime in the past
- Common temporal operators in Lustre: `Sofar`, `HasHappened`, `Since`

Basic temporal operators: talking about the past

- `Sofar(X)`:
  `X` has been true since startup of the program
- `HasHappened(X)`:
  `X` was true sometime since startup of the program
- `Since(X, Y)`:
  `X` was true sometime since startup of the program, and since then `Y` was true

Also common: operators to talk about the future (not possible in Lustre)

Further operator commonly used

- `RisingEdge(X)`:
  Value of `x` changes from `false` to `true`

Further temporal example

- In MultiStateSwitch:
  - if `pin2` is true and `pin0` is not released, `pin2` stays true

Safety vs. Liveness

- Different classes of requirements
- Safety:
  - **“Something bad never happens.”**
- Liveness:
  - **“Eventually, something good happens.”**
- **Synchronous observers can only express safety properties!**

How does Luke verify requirements?

Main techniques of Luke

- **Bounded model checking**
  - Constraint solving to detect error traces/counterexamples
  - Internally uses a SAT solver
  - Standard technique when designing hardware
- ***k*-Induction**
  - Strong form of mathematical induction
  - Prove that requirements hold

Bounded model checking

- Every Lustre program can be represented as a set of equations
- E.g.:

```
node Counter() returns (c : int);
let
  c = 0 -> (pre c + 1);
tel
```

↓

$$c_0 = 0$$
$$c_{i+1} = c_i + 1$$

Bounded model checking (2)

- We can unwind program/equations to generate counterexamples for properties
- Let's say, we try to prove for the counter that

  "*c* is always less than 10"

  (does not hold)

Bounded model checking (3)

- Generate *k* copies of the recurrence equations:

$$c_0 = 0$$
$$c_{i+1} = c_i + 1$$

→

$$c_0 = 0$$
$$c_1 = c_0 + 1$$
$$c_2 = c_1 + 1$$
$$c_3 = c_2 + 1$$
$$\vdots$$
$$c_{15} = c_{14} + 1$$

Bounded model checking (4)

- Check whether new equations imply property:

$$\begin{aligned} c_0 &= 0 \\ c_1 &= c_0 + 1 \\ c_2 &= c_1 + 1 \\ c_3 &= c_2 + 1 \\ &\vdots \\ c_{15} &= c_{14} + 1 \end{aligned} \quad \Longrightarrow \quad \begin{gathered} c_0 \leq 10 \wedge c_1 \leq 10 \\ \wedge \cdots \wedge \\ c_{15} \leq 10 \end{gathered}$$

- A SAT solver can check this quickly … and produce a counterexample

Bounded model checking (5)

- Bounded model checking can often show very quickly that some requirement **does not hold**

- **What if a requirement holds?**
    - Second technique in Lustre: *k*-induction

What is k-induction?

Imagine Fibonacci numbers ...

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_2 &= 1 \\
&\vdots \\
f_{i+2} &= f_i + f_{i+1}
\end{aligned}
$$

Let's prove that all Fibonacci numbers are non-negative:

$$\forall i.\ f_i \geq 0$$

$$f_0 = 0$$
$$f_1 = 1$$
$$f_2 = 1$$
$$\vdots$$
$$f_{i+2} = f_i + f_{i+1}$$

Proof using standard induction

- To show $\forall i.\ f_i \geq 0$
  we prove:
  - Base case: $f_0 \geq 0$
  - Step case: $f_i \geq 0 \Rightarrow f_{i+1} \geq 0$

- *Does not work for Fibonacci numbers*

Induction with two base cases *(2-induction)*

- To show $\forall i.\ f_i \geq 0$
  we can also prove:
    - Two base cases:

      $$f_0 \geq 0,\ \ f_1 \geq 0$$

    - "Simpler" step case:

      $$f_i \geq 0 \land f_{i+1} \geq 0 \ \Rightarrow\ f_{i+2} \geq 0$$

- *Works for Fibonacci numbers!*

k-Induction

- Generalises 2-induction to *k* base cases
- **Can be used to verify properties/requirements *P* of Lustre programs!**
    - **Base case:** prove that *P* holds in cycles 0, 1, 2, ..., (*k-1*)
    - **Step case:** assume that P holds in cycle *i*, *i*+1, *i*+2, ..., *i*+*k*-1, then prove that *P* also holds in cycle *i*+*k*

Non-inductive properties

- For some properties *P*, it can happen that step case fails, even though P always holds → *P* is **not inductive**

- E.g., $\forall i.\ f_i \geq 0$ is not inductive for *k*=1 (but for *k*=2)
- Some properties are not inductive for any *k*!

What to do in case of non-inductive properties?

- Method 1: **strengthen** the property P
  - verify not only P, but a stronger property **P & Q**

- Method 2: make the program to be verified more **defensive**
  - handle some cases that cannot actually occur
    → Luke might not be able to detect that the cases cannot occur

Summary of Luke V&V

- **Bounded model checking**
  - Used to show that some property **does not hold**
  - Generate a counterexample in this case

- ***k*-Induction**
  - Used to show that some property **always holds**

Further reading

- A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, 1999: “Symbolic Model Checking without BDDs”
- Sheeran, Singh, Stålmark, 2000: “Checking Safety Properties Using Induction and a SAT-Solver”

Equivalence checking using observers

- Synchronous observers can also be used to prove that two programs have the same behaviour
- E.g.

```
HasHappened(X) = not Sofar(not X)
```