Project 22703: SYRF

Deliverable 3.4.2

Deductive Proofs of Data-flow Programs

---

1 Introduction

This deliverable is intended to present the current year achievements with respect to task 3.4. This task aimed at exploring both the need of theorem proving for synchronous data-flow programs and ways to achieve it. As an example, the chosen data-flow language was LUSTRE and the theorem proving tool was PVS. According to the proposal, we had in mind the following schedule:

- deliverable 3.4.1 (12 months) : Comparative study of data-flow to PVS translations.
- deliverable 3.4.2 (24 months) : Case studies on checking data-flow programs with PVS.
- deliverable 3.4.3 (36 months) : Front-end prototype.

Yet, as explained in deliverable 3.4.1, some quite convincing progress on case studies were already performed in the first year [2], while some progress had still to be done concerning the choice of a translation scheme from LUSTRE to PVS. In between, we faced some bad experience with our first translation tool ([2] and D3.4.1). When we tried to present a demo of the tool (at TOOLS'98) we found that it was obsolete, because it couldn't run anymore on PVS new version 2.1 (cf. 3), and at the same time PVS old version 2.0 was not anymore supported on the available UltraSparc machine.

This unpleasant event led us to explore both a new translation method (section 2) and another way of implementing it (section 3).

2 Translation method

Our previous translation method, based on relations [2] was quite satisfactory, in that it didn't require neither compiling nor causality analysis, but exhibited two drawbacks:

- handling multiple clocks was uneasy: an attempt is proposed in [2] but limited to clocks having infinitely many occurrences,
- functional composition was quite awkward: because of the used relational approach, some intermediate variable was needed:

  In this approach, the LUSTRE node

```
node integr(x:real) returns (y:real);
     let y = x + (0.0 -> pre y) ; tel
```

is translated into the PVS theory:

```
integr[x,y : sequence[real]] : theory
  begin
    y_ax : AXIOM y = x + fby(0, y)
  end integr
```

Then instantiating twice the node in the equation:

```
z = integr(integr(x));
```

leads to the rather unpleasant translation:

```
yy, y : sequence[real]
IMPORTING integr[x, yy], integr[yy, y]
z_ax : AXIOM z = y
```

This led us to look for a functional approach not showing these drawbacks. It soon appeared that the one forecast in the preceding deliverable, based on modular co-iteration [3] could not satisfactorily handle recursive streams. We then investigated a semantic approach based on Kahn's theory of data-flow and linear Mu-calculus, and, up to now, this seems to solve most of our problems.

2.1 Kahn's theory

Kahn's theory [4] is based on the fact that the set of finite and infinite streams

$$D^\infty = D^* \cup D^\omega$$

is a CPO (complete partial order) with respect to the prefix order on streams:

$$x \leq y \Leftrightarrow \exists z : y = x.z$$

and the least element ϵ, (the empty stream).

In this setting, continuous functions of any order have least fixed points, and these fixed points also are continuous. Furthermore, it is easy to prove that every LUSTRE construct is continuous. Thus, a LUSTRE system of equation:

```
y = z + x;
z = integr(x);
```

will be translated into:

```
(y,z) = mu (lambda(y,z) : (z + x , integr(x)))
```

where + on sequences is also defined as a least fixed point:

```
map2(f) = mu (lambda(m):
              lambda(x,y): if x = empty or y = empty then empty
                           else cons(f(hd(x),hd(y)), m(tl(x),tl(y)))
                           endif
             )
conversion map2
```

Similarly, clock operators will be easily defined as fixed points. For instance:

```
when = mu (lambda(wh):
             lambda(c,x): if x = empty or c = empty then empty
                          else if hd(c)
                               then cons(hd(x), wh(tl(c),tl(x)))
                               else wh(tl(c),tl(x))
                               endif
                          endif
          )
```

2.2 Linear-time Mu-calculus

This theory closely matches linear-time Mu-calculus, where predicates can be defined as fixed points on the complete lattice of predicates. Furthermore, predicates can also be continuous, and when inverting the usual order on booleans ($t < f$), this will correspond to safety properties. Thus, if a predicate Q is continuous, we shall have the proof rule:

$$\frac{\begin{array}{l}\mathtt{Q}(\mathtt{empty}) \\ \mathtt{Q}(\mathtt{x}) \Rightarrow \mathtt{Q}(\mathtt{f}(\mathtt{x}))\end{array}}{\mathtt{Q}(\mathtt{mu}(\mathtt{f}))}$$

Due to the (usual) order inversion, recursively defined continuous predicates will correspond to greatest fixed points. For instance, the All(P) predicate on streams will be defined as:

```
All(P) = Nu( lambda(X): First(P) And Next(X))
```

with the semantics:

```
First(P)(x) = if x = empty then true
              else P(hd(x))
              endif

Next(X)(x)  = if x = empty then true
              else X(tl(x))
              endif
```

Here, `Next` is the usual modality of linear-time temporal logic, but for the `empty` case. Yet we can notice the need for the unusual `First` modality, which is caused by the fact that, mainly due to clocks, we need to handle finite streams as well:

In usual temporal logic, a predicate on values is implicitly extended to operate on infinite streams:

```
P(x) iff  P(hd(x))
```

This cannot work anymore on possibly empty streams, if we want `P` to be continuous, because we should require the contradiction:

```
P(empty) And Not(P(empty))
```

Adding the `First` modality solves the problem because now, `Not(First(P))` and `First(Not(P))` are not the same[1].

3 Implementation

The previous translator was implemented using "hidden" PVS primitives, and it appeared that some of them were changed when moving from 2.0 to 2.1 version. Since these primitives were not public, the PVS team cannot be blamed for it. Yet, as mentioned above, our prototype could not run anymore. This bad experience led us to think that it would be safer to only use "public" primitives.

Thus we designed a (very simple) translator from LUSTRE to PVS based on the scheme presented above, and we are now working on defining proof strategies for (mainly safety) properties of programs, which will be implemented as (predefined) PVS proof files (`.prf` files).

4 Conclusion

Both practical and theoretical reasons led us to rework our translation scheme from LUSTRE to PVS. Our first experiments lead us to think that this scheme is convenient:

- it easily allows the translation of LUSTRE features in a natural way,
- it is even more general, allowing the translation of any asynchronous and higher order Kahn network,
- it does not need misusing some useful PVS features such as subtyping,
- it retains the traditional two stage program proof method:
  1. partial correctness,
  2. termination.

  instantiated here as:

[1] It seems that this was already noticed by the authors of LUCID, (a LUSTRE ancestor) [1]. It may be that the history of temporal logic in computer science has not yet been exhausted.

1. partial correctness,
2. deadlock freedom (also called productivity).

Furthermore, in the LUSTRE case, this later step can be automatically handled by the LUSTRE compiler.

We plan to devote the next year to implementing this scheme.

References

[1] E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3:336–354, 1976.

[2] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999. to appear.

[3] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. In *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, Lisbon*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1998.

[4] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.