Project 22703: SYRF

Deliverable 3.4.1

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 this schedule was rather theoretical: for instance, in order to compare translations, we needed some case studies. Similarly, in order to have thorough comparisons we also needed some assistance from automated tools.

This led us to progress in each direction at the same time:

- We designed a translation tool based on a simple translation principle and used it to treat examples. This tool was also made available on the web at location: `http://www.imag.fr/VERIMAG/SYNCHRONE/PVS/` so as to be used by other partners in the project. This tool is briefly described in section 2.
- We designed some examples, which we felt typical of problems that cannot be model-checked. These examples are further described in section 3.
- Based on these examples we made some comparison between alternative translation methods. At the same time, we launched some study of other methods (section 4).

2 A translation tool

The problem of translating data-flow programs to the language of the PVS prover is the following one: on the one hand, data-flow programs are mainly stream programs that can be dealt with either in a lazy fashion or in a compiled (co-iterative) way [1, 2]; on the other hand, the PVS language is a strict functional language which is not consistent with laziness [4].

Thus, it seems at first glance that a good solution would be to base the translation on the compiled code, and this is the solution taken by several researchers in the field [3, 5]. Yet, we felt that it would be better to base the translation on the source code, for the reason that programmers are intended to be end-users of the tool; if they choose data-flow programming, it is because they think of their programs in terms of data-flow and similarly, want to check them within the same framework. Furthermore, basing the translation on the source code could appear less demanding when it comes to comparisons.

This led us to adopt the following translation principle:

- translate streams as functions from naturals,
- deal with recursive stream programs in a relational way, as it cannot be done functionally because PVS does not accept it,
- and use the compiler to check that translated programs are well-behaved with respect to causality (recursion): thus the above described relations have an unique solution and are in fact functions.

More details on the translation tool, its description, user manual and application examples can be found on the web at the above given location.

3 A typical example: a toy railway control problem

This section is intended to provide a simple, small and yet significant example showing the interest of our approach. It is based on a toy problem consisting of controlling automatically a railway system. The idea is the following: each train on the track has a computerized controller which receives periodically some information about its speed, given by some sensor, and the speed (as a matter of fact, the motion within a time unit) of the train in front of him, broadcast by some transmission device, and has to actuate an emergency brake in order to avoid colliding with this train.

3.1 Specification

In this specification, all variables are LUSTRE variables and denote streams. Our specification is made, first, of assumptions about the physical world. For this, we consider the train B that is behind:

- Its acceleration, `a`, is, if the controller actuates the brake then `-afu`, where `afu` is the (positive) deceleration of the emergency brake, otherwise `am`, some unknown stream, yet not larger than a maximal acceleration constant `amax`:

```
const afu, amax;
a = if fu then -afu else am;
assert( (afu > 0 ) and (am < amax) );
```

- Trains are not supposed to move backward and are assumed to be initially stopped. Furthermore, we assume streams sampled with a time unit period. Thus the speed of train B, `v`, is its preceding speed plus the acceleration, `a`, if positive, else `0`.

```
vv = (0 ->pre v) + a ;
v  =  if vv>=0 then vv else 0;
```

- The distance `d` between trains A and B is the preceding distance plus `vp`, the speed of train A, minus `v`, the speed of train B. A positive initial distance is assumed:

```
const dinit;
d = (dinit ->pre d) + vp - v;
assert((dinit > 0 ) and (vp >= 0));
```

Given this environment a specification could be:

Find a controller law for `fu` *such that trains never collide:* `d > 0`

3.2 A controller proposal

`fu` is a boolean stream which says whether the train has to brake or not. To synthesize it, at each time, the worst situation has to be considered : the train A stops (its speed is 0) and train B has the maximal acceleration `amax`. This allows the estimatimation of the maximal speed and thus the minimal distance that can be reached. This minimal estimated distance has to be positive if we do not want the two trains to collide. But this is clearly not enough and the controller has to perform some forecast in order to check whether it will be able to stop the train in due time in the future. For the sake of simplicity, we choose here a one time unit forecast: if it chooses not to brake now, and if it takes a maximal acceleration, it must be able to stop at next time unit. The boolean value `fu` is then true if the train B must brake now (i.e. if the minimal estimated distance is negative or if, in the worst case it will not stop, though braking at next instant) and false otherwise.

```
fu = ((dinit ->pre d) - (0 ->pre v) - amax < 0)
     or ((0 ->pre v) + amax - afu >=0);
```

3.3 Controller correctness

In order to prove the controller correctness, we translate the specification into PVS. This translation is shown at table 1.

Then, we prove that the two trains will never collide.

```
correctness : THEOREM assert(map2(>)(d, const(0)))
```

The proof fails first because we need some assumption to limitate the maximal train acceleration:

```
accelerationmax : ASSUMPTION  assert(map2(>)(afu,amax))
```

```
train: THEORY
 BEGIN
   IMPORTING lustre[bool], lustre[int], map[int,int],
             map2[bool,bool,bool], map2[int,int,int],
             map2[int,int,bool]

    am,vp,a,v,d : sequence[int]
    fu : sequence[bool]

    amax,afu,dinit : sequence[int]
    amaxval, afuval, dinitval : int

    amax_val : ASSUMPTION amax = const(amaxval)
    afu_val : ASSUMPTION afu = const(afuval)
    dinit_val : ASSUMPTION dinit = const(dinitval)

    acceleration : ASSUMPTION a = ifl(fu,map(-)(afu),am)

    acceleration_laws : ASSUMPTION assert(map2(>=)(afu,const(0)))
                              AND assert(map2(<)(am,amax))

    speed : ASSUMPTION v =
                      let vv = map2(+)(pre(0,v),a)
                      in  ifl(map2(>=)(vv,const(0)),vv,const(0))

    distance : ASSUMPTION d = map2(+)(fby(dinit,d),map2(-)(vp,v))

    distance_laws : ASSUMPTION assert(map2(>)(dinit,const(0)))
                          AND assert(map2(>=)(vp,const(0)))

    braking : ASSUMPTION fu = map2(OR)
                           (map2(<)(map2(-)(fby(dinit,d),
                                            map2(+)(pre(0,v),
                                                    amax)),
                                    const(0)),
                            map2(>=)(map2(+)(pre(0,v),amax),afu))
  END train
```

Table 1: Train theory

We then try to prove the theorem by induction on naturals[1] since all Lustre variables are sequences indexed by naturals. The proof fails because it requires a preliminary lemma:

```
speedmax: LEMMA  assert(map2(<=)(v,afu))
```

This lemma is easily proved by induction on naturals. It just needs rewriting the Lustre equations in the right order, some case analysis and using the given assumptions.

The proof of the main theorem follows immediately by induction on naturals based on the same principle.

3.4 A more realistic controller

A more realistic controller is based on a final distance function `df(d,v)` that is the distance between the current train and the preceding one that will be reached starting from distance `d` and speed `v` if the later doesn't move and if the former brakes continuously with deceleration `-a`. It is mathematically defined by:

$$df(d,v) = d - \sum_{i=0}^{u}(v - \sum_{j=0}^{i} a) = d - uv + \frac{u(u+1)}{2}a$$

where $u = floor(\frac{v}{a})$ is the number of steps up to which the speed will become zero.

The controller law amounts to saying that, assuming the current train takes a maximal acceleration now and begins braking only at next step, if the final distance is negative then the train must brake now.

The following properties have then be proved in PVS

- speed is always positive,
- a positive final distance is a system invariant
- as a consequence, trains will not collide.

The proof is 90 pages long and took one week to be completed.

Other examples can be found in the release on the web.

4 Other approaches

4.1 A comparison with co-iteration

A hand-made translation based on compiled code has been performed on the toy train example, and proofs have been completed. This allowed us to notice that such a translation is as good as the previous one and even better in the sense that it works faster: this is due to the fact that now the translation is both strict and functional and in PVS, function application works faster than relation replacement.

However, this conclusion may be moderated by noticing that such example is too small for benefiting from modular reasoning.

[1] i.e. $P(0) \wedge \forall n \in \mathcal{N}.(P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathcal{N}.P(n)$

4.2 Modular co-iteration

In a recent work, [2] we found a set of combinators for compiling (higher order and recursive) stream programs in a co-iterative form. This may allow the definition of a kind of modular LUSTRE compiler, directly in PVS. If this works, we could directly translate source code into compiler input and directly perform proofs on this translation, without applying (beta-reducing) the combinators.

5 Conclusion

A prototype translator has been implemented and case studies have been run on it in a seemingly satisfactory way. Also a comparison with another method has been achieved and another alternative translation method has been studied.

The main lesson that can be drawn from this experiment is the following: in a preliminary remark we assumed that users wanted to stay as near as possible from their initial data-flow formalism. If this is true, both performed translations fail in the sense that, if a given property is not automatically proved, the user has to deal with it in the PVS language and not in the source language. This leads us to think that the real problem is not the particular chosen translation method, but an inverse translation allowing prover sequents to be expressed in terms of the source language.

We therefore propose to investigate this question in the coming year, and then to design a front-end prototype during the last year.

References

[1] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. on Functional Programming*. ACM SIGPLAN, Philadelphia May 1996.

[2] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. Research report, Verimag, October 1997. available at `http://www.cadillac.ibp.fr/p̃ouzet`.

[3] P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[4] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992.

[5] C. Paulin-Mohring. Circuits as streams in Coq, verification of a sequential multiplier. Research Report 95-16, Laboratoire de l'Informatique du Parallélisme, September 1995. Available at `http://www.ens-lyon.fr:80/LIP/lip/publis/`.