Project 22703: SYRF

Deliverable 3.3.3

Handling properties depending on numerical variables

The principles of our verification tool were described in the deliverable 3.3.2. These previous results were published in [JHR99]. This year, we first give a brief description of the new developments, and then we give some experimental results.

1 Progress

This year, task 3.3 mainly progressed in two directions:

- Algorithms were significantly improved, both concerning efficiency and precision of the results. In particular a new library for handling convex polyhedra was written, and the technique for computing pre- and post- conditions was improved.

- The heuristics to perform location splitting were also improved and tuned.

1.1 Algorithmic improvements

New polyhedra library: The code we used had a main limitation: rational coefficients of constraints and vertices were represented with machine integers (usually 32 bits integers). This led to overflow problems when trying to use our verification tool even on a small example with 6 numerical variables.

As a consequence, the first motivation was to be able to choose the representation of integers. Now you can choose between 32 bits integers, 64 bits integers, and multi-precision integers from the GNU Multi-Precision (GMP) library.

Furthermore, we incorporated some simple techniques described in [FP96] and optimized the upper layer of the library. For instance, we can now keep in memory some intermediate structures that avoid to perform several times the same computations; this results in a more clever use of the core and costly algorithm which is incremental.

Computation of pre- and post-conditions: As explained in the deliverable 3.3.2, we use a sequential version of the transition function to perform these computations. In the general case, this technique speeds up the computation, but is less precise in the results. We designed a technique based on a combination of forward/backward computations to improve the precision of these operations at low cost.

1.2 Refinement heuristics

The principle of the refinement is to split locations according to conditions which separate different behaviors from these locations. In general, there are many locations that can be split, and each of them can be split according to many conditions. However, we want to avoid, as much as possible, useless splitting. For instance, approximations which prevents the tool from verifying the property can take place at a high level, between strongly connected components of the control structure, or in deep internal loops. As a consequence, to express and to control the splitting strategy, we use Bourdoncle's algorithm to decompose the control structure into Strongly Connected Sub-Components [Bou92].

2 Experimental results

In this whole section, times results are those obtained with the `time` command, on a computer equiped with 512 Mb of memory and a Pentium II processor running at 450 MHz. Memory use, very rougly measured by running the `top` command, never exceeds 20 Mb. The kind of integer representation used by the polyhedra library is precised.

2.1 A toy example: alternate counters

j

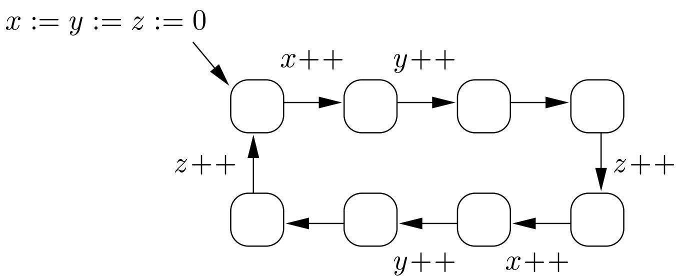

Figure 1: Alternate counters, with $n = 2$

This example is a ring of locations, as described on the figure 1. The program performs n times the same sequence of operations, before coming back in the initial location. It is obvious to see that this sequence preserves the invariant $x \geq y \geq z$, which is initially satisfied. In order to check this property, we add an auxiliary boolean variable defined by

```
ok' = ok and (x >= y) and (y >= z)
```

where `ok` is initially true, and `ok'` is the value of `ok` in the next step. We want to verify that `ok` cannot become false.

Tables 2.1 and 2 shows experimental results. **n** is the number of repetition of the basic sequence, **locations** indicates what would be the size of the expanded boolean control automaton, **steps** is the number of forward/backward analysis needed, **splits** indicates how many times we have split a location (into two locations), and **time** gives the execution time depending of the type of integers used in the polyhedra library. Notice that the numbers of steps and splits do not change. The execution time increases with n because of the boolean propagation in forward and backward analysis, and because of the ring structure of the boolean control automaton.

n	locations	steps	splits	time(sec)		
				32 bits	64 bits	GMP
1	4	5	5	0.1	0.1	0.2
2	8	5	5	0.2	0.1	0.3
4	16	5	5	0.2	0.2	0.5
16	64	5	5	0.6	0.7	1.4
256	1024	5	5	18.8	19.9	29.7

Table 1: Verification of the property `ok` with `ok' = ok and (x >= y) and (y >= z)`

n	locations	steps	splits	time(sec)		
				32 bits	64 bits	GMP
1	4	2	1	< 0.05	< 0.05	< 0.1
4	16	2	1	< 0.1	< 0.1	< 0.2
16	64	2	1	0.3	0.3	0.4
256	1024	2	1	12.1	12.7	15.5

The verification of this simpler property is much easier in term of number of splits: it is enough to split the initial invariant location according to $x \geq y$. This shows that the verification steps depend effectively of the property to be proved.

Table 2: Verification of the property `ok` with `ok' = ok and (x >= y)`

2.2 A simple lift system

As a more complex example, we consider a (very simple) lift system, roughly represented in figure 2. The lift connects two different floors, and has two input commands: going down and going up. Before reacting to an input command, the controller first checks that the lift is free and stands at the right floor. If so, it closes the door, then, depending of the received input command, it goes up or down and opens the door at the new floor. Closing and opening the door, as well as going up and down, take a certain amount of time; as a consequence a counter is associated with each of these operations to model the delays. Reaction to commands is often delayed, which increases the number of boolean state variables.

The type of properties we wish to prove are mainly delay properties.

Property 1 We want to check an upper bound for the delay between the beginning of the door closing and the end of the door opening.

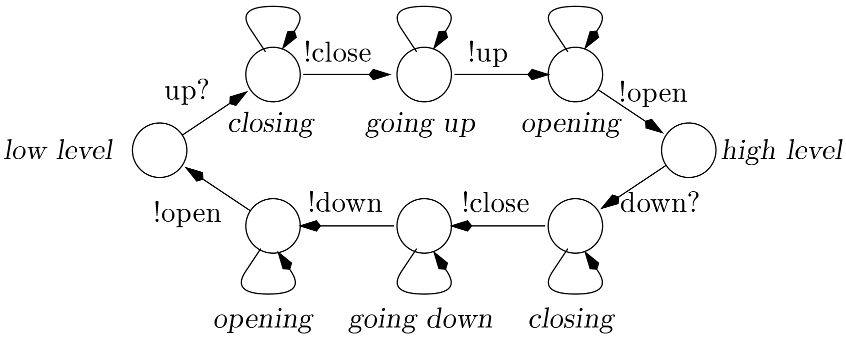

Figure 2: Lift system

Property 2 We want to check a lower bound for the delay between the beginning of the door closing and the end of the door opening.

Property 3 We try to check a lower bound between two moves up; this implies that the lift moves also down once in between.

Property 4 is a coherence property: we check that the lift is at most in one of these states: opening the door, closing the door, going up, and going down.

Table 3 gives experimental results on the verification of these properties. In the subcolumns of the column **var and cond**, **bool** gives the number of boolean state variables, **num** the number of numerical state variables, and **cons** the number of constraints appearing in the source of the program. In the subcolumns of column **locations**, **initial** indicates what would be the size of the expanded boolean control automaton, and **analysis** gives the same information, if we build the control automaton after a boolean reachability analysis taking into account the unsatisfiability of some numerical guards. The others columns have the same meaning as in the previous tables.

prop	var & cond			locations		steps	splits	time(sec)		
	bool	num	cons	initial	analysis			32/64/GMP		
1	12	6	9	3072	2062	8	20	10.1	11.4	25.9
2	12	6	9	3072	2062	7	15	9.7	11.1	25.5
1 & 2	12	6	10	3072	2062	8	20	12.9	14.7	33.4
3 (failed)	12	6	9	3072	2062	9	22	12.0	13.4	26.0
4	12	4	8	3072	0	–	–	0.09	0.12	0.20

Table 3: Simple lift system

We give here some comments on this results.

1. The influence of the type of integers used in the polyhedra library is more apparent here. The use of 64 bits integers instead of 32 bits ones increases the verification time of approximately 10 %, and the use of multi-precision integers increases it by a factor of more than two.

2. For properties 1,2 and 3, normally we should have only 5 numerical variables, but the present LUSTRE compiler doesn't detect the identity of two counters when expanding the nodes. Of course, the verification tool detects it with the first forward analysis.

3. We were not able to prove property 3. The reason is that the counter associated with the door opening, for example, is used twice: after the first move up, and after the move down. In order to be able to prove the property, the tool should distinguish these two phases, to compute the delay of two door opening. The same observation holds for the counter associated with the door closing.

   In some sens, this property can be considered as non linear, even if it is syntaxically linear: it depends of the comparison of the bound with the multiplication of two constants appearing in the program (2 and the delay for the door opening).

4. Property 4 is proved only by removing unfeasible transitions in a boolean reachability analysis.

References

[Bou92] F. Bourdoncle. Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite. Thesis, Ecole Polytechnique, Paris, 1992.

[FP96] K. Fukuda and A. Prodon. Double description method revisited. In *Combinatorics and Computer Science,*, volume 1120 of *LNCS*, pages 91–111. Springer-Verlag, 1996.

[JHR99] B. Jeannet, N. Halbwachs and P. Raymond. Dynamic Partitioning in Analyses of Numerical Properties. In *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.