Project 22703: SYRF

Deliverable 3.3.2

Handling Numerical Properties

---

1 Introduction

In deliverable 3.3.1, Section 4, we described the main goal of this task: linear relation analysis — to verify properties depending on numerical variables — must be improved, in particular in the choice of a suitable control structure. On one hand, the control structure deduced from the exhaustive simulation of the behavior of Boolean variables is generally too large to be used as a basis for linear relation analysis, and, on the other hand, it is sometimes too coarse to permit the verification of the given property. Our goal is to deduce the suitable control structure from the property to be proved, by progressively refining a coarse control structure.

Such an approach is necessarily based on heuristics, which can only be adjusted from experiments. Our efforts, this year, have been essentially devoted to the design of an experimental workbench, to play with such heuristics. A simple format has been defined (Section 2), to symbolically describe interpreted automata, allowing arbitrary refinement of the control structure (Section 3). Linear relation analysis has been adapted to this format (Section 4) and some refinement strategies have already be defined (Section 5).

2 Symbolic Representation of Interpreted Automata

2.1 Syntax

We use a very simple format to describe automata: An interpreted automaton is given by

- the declaration of Boolean state variables $(b_i)_{i=1\ldots m}$ and input variables $(c_j)_{j=1\ldots n}$
- the declaration of numerical[1] state variables $(x_k)_{k=1\ldots p}$ and input variables $(y_\ell)_{\ell=1\ldots q}$
- a set of equations

$$\left\{ \begin{array}{lcll} b'_i & = & \phi_i(\vec{b},\vec{c},\vec{x},\vec{y}) & i=1\ldots m \\ x'_k & = & \psi_k(\vec{b},\vec{c},\vec{x},\vec{y}) & k=1\ldots p \end{array} \right.$$

giving the value of each state variable at the next instant, as a function of the current values of state and input variables. In these equations, expressions ϕ_i and ψ_k are

[1] For the time being, numerical variables are integers, but handling rational variables is easy.

well-formed expressions (of suitable types), freely mixing Boolean expressions and *linear* numerical expressions.

- Two Boolean functions $Init(\vec{b}, \vec{x})$ and $Inv(\vec{b}, \vec{x})$, respectively defining the set of initial states and the invariant (i.e., the property to be verified) of the automaton.

- possibly an assertion $Assert(\vec{b}, \vec{c}, \vec{x}, \vec{y})$, which defines, as in LUSTRE, a restriction on the possible inputs to the automaton.

Example: We describe a system which can be either "*running*" or "*not running*". Initially, it is "*not running*". It enters the "*running*" state when the Boolean input d is true, and stays in this state for 4 steps, as specified by the counter n. An assertion specifies — by means of another counter m — that the input d is true at most once every 6 steps. The property to be verified is that d is never true when the system is "*running*"; it is specified by means of another state variable "*bad*", since the invariant can only be defined by means of state variables:

```
state       running, bad;
num_state   n, m;
inputs      d;
initial     not running and not bad;
transition
            running' = d or (running and n < 4);
            n' = if running then n+1 else 0;
            m' = if d then 0 else m+1;
            bad' = bad or (running and d);
assert      d => (m >= 6);
invariant   not bad;
```

Notice that, although this example could be described by a (discrete) time automaton [AD90], our model is much more powerful. It corresponds, in fact, to a discrete and deterministic version of *linear hybrid automata* [ACH⁺95].

2.2 Semantics and Verification Goal

A *state* of the automaton is a pair $(\vec{\beta}, \vec{\xi}) \in \mathbb{B}^m \times \mathbb{Z}^p$ of valuations of the Boolean and numerical state variables. A state $(\vec{\beta}, \vec{\xi})$ is initial iff $Init(\vec{\beta}, \vec{\xi}) = true$. The transition relation is defined by

$$(\vec{\beta}, \vec{\xi}) \xrightarrow{(\vec{\chi}, \vec{\nu})} (\vec{\beta}', \vec{\xi}') \iff \begin{cases} \beta_i' = \phi_i(\vec{\beta}, \vec{\chi}, \vec{\xi}, \vec{\nu}) & i = 1 \ldots m \\ \xi_k' = \psi_k(\vec{\beta}, \vec{\chi}, \vec{\xi}, \vec{\nu}) & k = 1 \ldots p \end{cases}$$

A run of the automaton is an infinite sequence $(\vec{\beta}^{(i)}, \vec{\chi}^{(i)}, \vec{\xi}^{(i)}, \vec{\nu}^{(i)})_{i \geq 0}$, where

- $(\vec{\beta}^{(0)}, \vec{\xi}^{(0)})$ is an initial state;

- for all $i \geq 0$, $(\vec{\beta}^{(i)}, \vec{\xi}^{(i)}) \xrightarrow{(\vec{\chi}^{(i)}, \vec{\nu}^{(i)})} (\vec{\beta}^{(i+1)}, \vec{\xi}^{(i+1)})$

- for all $i \geq 0$, $Assert(\vec{\beta}^{(i)}, \vec{\chi}^{(i)}, \vec{\xi}^{(i)}, \vec{\nu}^{(i)}) = true$.

A state $(\vec{\beta}, \vec{\xi})$ is reachable if it belongs to the projection of a run onto state variables. The goal of the verification is to show that, for any reachable state $(\vec{\beta}, \vec{\xi})$, $Inv(\vec{\beta}, \vec{\xi}) = true$.

3 Control Structures and Abstractions

3.1 Control Structures

Let $S = \mathbb{B}^m \times \mathbb{Z}^p$ be the set of states of a symbolic automaton. A *control structure* is a finite covering $C = \{s_i \mid i = 1 \ldots r\}$ of S (i.e., a finite set of subsets of S such that $\bigcup_{i=1}^{r} s_i = S$). The *abstraction* of the symbolic automaton according to C is the finite automaton $(C, C_0, \rightarrow)$ such that

- the set C_0 of initial states is made of all the s_i that contain at least an initial state of the symbolic automaton;
- $s_i \rightarrow s_j$ iff there exists $(\vec{\beta}, \vec{\xi}) \in s_i$, $(\vec{\beta}', \vec{\xi}') \in s_j$, and there exists $(\vec{\chi}, \vec{\nu})$, such that $(\vec{\beta}, \vec{\xi}) \xrightarrow{(\chi,\nu)} (\vec{\beta}', \vec{\xi}')$ and $Assert(\vec{\beta}, \vec{\chi}, \vec{\xi}, \vec{\nu}) = true$.

3.2 Convex States

In this work, we will restrict ourselves to control structures, each element s_i of which is a Cartesian product $B \times P$, where B is an arbitrary subset of $\mathbb{B}^m$ (described by a Boolean formula, or a BDD) and P is a convex polyhedron of $\mathbb{Z}^p$ (represented by a system of linear constraints). We call such a pair "*a convex state*". Convex states are closed by intersection:

$$(B_1, P_1) \cap (B_2, P_2) = (B_1 \wedge B_2, P_1 \cap P_2)$$

but not by union, which is replaced by *convex closure*:

$$(B_1, P_1) \sqcup (B_2, P_2) = (B_1 \vee B_2, P_1 \sqcup P_2)$$

where $P_1 \sqcup P_2$ denotes the convex hull of polyhedra.

3.3 Initial Abstraction

A trivial abstraction consists in choosing $C = \{(\mathbb{B}^m, \mathbb{Z}^p)\}$, but this coarsest abstraction is generally of little interest.

Obviously, if $s_i \in C$ is not reachable (from an initial state $\in C_0$) in the abstraction, no concrete state $(\vec{\beta}, \vec{\xi}) \in s_i$ is reachable in the symbolic automaton[2]. So, since our goal is to show that no concrete state violating the invariant can be reached in the symbolic automaton, it is reasonable to distinguish states violating the invariant. This provides us with a first control structure:

- if $\neg Inv$ is a convex state, take it and the convex closure of Inv as initial control structure;

[2]This property allows us to deal with state *covering* instead of *partitionning*.

- otherwise, take the convex closure of *Inv*, together with a coverage of $\neg Inv$ by convex states. The reason for detailing the complement of the invariant (instead of the invariant itself) is that the possible explosion of abstract states it involves will not be expensive (erroneous states are considered as sinks, and their future is not analyzed).

Moreover, since we are looking for paths from initial states to states violating the invariant, *Init* and *Inv* play a symmetrical role. As a consequence, it is also reasonable to refine the control structure (in the same way as above) according to initial states.

Example of decomposition: let us consider the set of states defined by $(b \equiv x > 0)$. Its convex closure is $\mathbb{B} \times \mathbb{Z}$, so it loses all information. Now, we can decompose it according to the value of b, getting two convex states $(b \wedge x > 0)$ and $(\neg b \wedge x \leq 0)$; in this case we don't lose any information.

4 Computation of Post- and Pre-Conditions

The next step is to compute an upper approximation of the relation transition over abstract states. For that, we apply linear relation analysis to our abstract automata. Each abstract state s has a definition def_s — of the form $B \times P$, defining the set of concrete states it represents — , and will be provided with two other attributes, also decomposed as Cartesian products of a Boolean formula and a polyhedron:

- The set $reach_s$ will be an upper approximation of the set of concrete states in s which are reachable in the symbolic automaton. This set will be computed by a forward analysis starting from abstract states that are known to contain initial concrete states.

- The set $danger_s$ will be an upper approximation of the set of concrete states in s which can lead, in the symbolic automaton, to a state violating the invariant. This set will be computed by a backward analysis from abstract states that are known to contain concrete states violating the invariant.

With these notations, our verification goal is to show that $\forall s \in C, reach_s \cap danger_s = \emptyset$.

4.1 Forward Analysis, Post-conditions, and Transition Sequentialisation

In a forward analysis, we have to compute the postcondition of an abstract state s, i.e,

$$post_s = \{(\vec{\beta}', \vec{\xi}') \mid \exists(\vec{\beta}, \vec{\xi}) \in reach_s, \exists(\vec{\chi}, \vec{\nu}) \text{ such that } (\vec{\beta}, \vec{\xi}) \xrightarrow{(\chi,\nu)} (\vec{\beta}', \vec{\xi}')\}$$

To know if there can be a transition from s to s' (according to the current state of the analysis), we just have to check whether $post_s$ intersects $def_{s'}$.

4.1.1 Post-conditions

Let $reach_s = (B(\vec{\beta}) \wedge P(\vec{\xi})$. To compute $post_s$, we can consider the formula

$$reach_s \ \wedge \ (\vec{\beta},\vec{\xi}) \xrightarrow{(\chi,\nu)} (\vec{\beta}',\vec{\xi}')\}$$

and project it, on one hand onto $\vec{\beta}'$ — to get the Boolean part of *post* —, and on the other hand onto $\vec{\xi}'$, with suitable convex hulls — to get its polyhedral part. This leads to a very rough result, because the Boolean and numerical parts of the transition function are considered independently. A more precise solution consists in decomposing the formula into subformulas, such that, in each subformula, the relations linking numerical variables be *linear*.

For instance, let $reach_s = ((b \vee c) \wedge x > 0)$, and assume that the transition function is

```
x' = if b then x+1 else x;
b' = not b;
c' = b or c;
```

In the first case, we project the formula

$$(b \vee c) \wedge x > 0 \wedge (b \Rightarrow x' = x + 1) \wedge (\neg b \Rightarrow x' = x) \wedge b' = \neg b \wedge c' = (b \vee c)$$

onto b', c', getting c', and onto x, getting $x > 0$. In the second case, the formula is split (according to b or $\neg b$) into

$$b \wedge (b \vee c) \wedge x' = x + 1 \wedge b' = \neg b \wedge c' = (b \vee c)$$

which is projected onto $\neg b' \wedge c' \wedge x' > 1$, and

$$\neg b \wedge (b \vee c) \wedge x' = x \wedge b' = \neg b \wedge c' = (b \vee c)$$

which gives $b' \wedge c' \wedge x' > 0$.

However, both cases present the same drawback, which is that we have to deal with formulas involving both unprimed and primed variables. Increasing in that way the number of variables in formulas is expensive in BDD computation, and prohibitive in polyhedra.

This is why we prefer considering a sequential implementation of the transition function, similar to the one that is built by a compiler [Ray91]: variables are sorted according to their dependences, only a reduced number of auxiliary variables ("buffers") are introduced when necessary, and the postcondition is computed progressively along the resulting sequence of assignments. In the previous example, the sequential version of the transition function would be

```
if b then x:= x+1;
c := b or c;
b := not b;
```

and the progressive computation of the postcondition gives

$\boxed{((b \vee c) \wedge x > 0)}$
if b then x:= x+1; $\boxed{((b \vee c) \wedge x > 0)}$
c := b or c; $\boxed{(c \wedge x > 0)}$
b := not b; $\boxed{(c \wedge x > 0)}$

and we get the same result as in the first case above, without dealing with more variables. The second case corresponds to a better factorization of conditional, considering the sequential code

```
if b then
    x:= x+1;
    c := b or c;
    b := not b;
else
    c := b or c;
    b := not b;
```

on which the computation gives

$\boxed{((b \vee c) \wedge x > 0)}$
if b then $\boxed{(b \wedge x > 0)}$
    x:= x+1; $\boxed{(b \wedge x > 1)}$
    c := b or c; $\boxed{(b \wedge c \wedge x > 1)}$
    b := not b; $\boxed{(\neg b \wedge c \wedge x > 1)}$
else $\boxed{(\neg b \wedge c \wedge x > 0)}$
    c := b or c; $\boxed{(\neg b \wedge c \wedge x > 0)}$
    b := not b; $\boxed{(b \wedge c \wedge x > 0)}$

4.1.2 Transition Sequentialisation

Before computing a postcondition, our analyzer will produce such a sequential version of the transition function. Moreover, it will play with the structure of this sequential code: a tree-like structure, like in the last version above will give more precise results, but may involve an explosion of the size of the code, and thus, of the computation cost. A transition sequentialisation consists of three steps:

1. **analysis of dependencies over state variables:** we build the dependence graph deduced from the equations. In this graph, $x \to y$ means that the current value of y is used to compute the next value of x, and that we should update x before y.

2. **determination of buffers, and scheduling:** the dependance graph may contain loops, e.g., if the computation of x' and y' respectively depend on y and x:

   ```
   x' = f(y); y' = g(x);
   ```

   These loops must be broken, by introducing buffers. For instance, for the above example, a correct assignment sequence could be:

   ```
   buff := x; x = f(y); y = g(buff);
   ```

   Finding a minimum number of buffers is NP-complete. We use Bourdoncle's "strongly connected subcomponents" algorithm [Bou92] as a heuristic to introduce a reasonable (most often, minimal) number of buffers.

3. **test factorization:** the last problem is to choose a strategy to open and close test statements (if-then-else-). As said before, this choice is a compromize between the cost and the precision of the analysis. For instance, consider the sequence

```
b1 := if (y < 3) then (y > 0) else (x1> 0);
x1 := if b2 then x1+1 else 2*x1;
b2 := if (y < 3) then (y > 0) else true;
```

If we accept the duplication of the assignment to x1, we can factorize the test on y < 3, getting

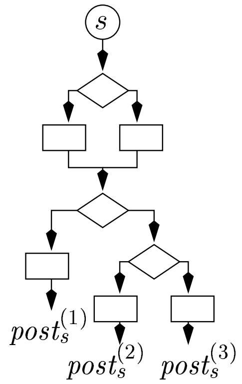
```
if (y < 3) then
    b1 := (y > 0); x1 := if b2 then x1+1 else 2*x1; b2 := (y > 0);
else
    b1 := (x1> 0); x1 := if b2 then x1+1 else 2*x1; b2 := true;
```

which is likely to provide more precise results (e.g., $b_1 \Rightarrow b_2$).

According to our general philosophy, we start with a cheap computation, but we will be free (see §5.2) of re-openning a test if this appears to be useful in the verification. To start with, we apply the following heuristics: after openning a test (if-then-else-) we schedule in each branch all the computations that depend on the condition of the test and that can be scheduled according to the dependence relation. After that, the test is closed, except if it is the last one.

4.1.3 Forward Analysis

Since the last tests are not closed, in general, the postcondition $post_s$ will be the disjunction of all the partial postconditions $(post_s^{(i)})_i$ computed on the branches of these last tests (see the Figure besides). A branch can lead to an abstract state s' only if the corresponding partial postcondition intersects $def_{s'}$. The convex closure of these intersections over all branches provides the postcondition $post_{s,s'}$, and the convex closure $\bigsqcup_s post_{s,s'}$ gives the new approximation $reach_{s'}$. The forward analysis takes abstract states into account as they become (apparently) reachable. The computation is iterated in a standard way until convergence, with extrapolation at "widening states" chosen again by means of Bourdoncle's "strongly connected subcomponents" algorithm [Bou92].



4.2 Backward Analysis

The goal of the backward analysis is to compute, in each abstract state s, the subset $danger_s$ of the concrete states in s which can lead to the violation of the property. Of course, only those of these concrete states that are reachable are really "dangerous", so the results of the backward analysis are always intersected with $reach_s$. If $danger_{s'}$ is not empty, its precondition over an other state s is computed by a backward propagation:

for each branch of the transition from s whose partial postcondition $post_s^{(i)}$ intersects $danger_{s'}$, the condition $post_s^{(i)} \wedge danger_{s'}$ is propagated backward to s; the combination of the results of this backward propagation over all branches gives the precondition $pre_{s',s}$, and the convex closure $\bigsqcup_{s'} pre_{s',s}$ provides a new solution for $danger_s$. Of course, this computation is iterated, with extrapolation, until convergence.

More generally, forward and backward analyses are performed several times in alternance, until convergence, replacing, at each step, each state definition def_s by $reach_s \wedge danger_s$. Whenever def_s is strengthenned, the transition sourced in s can be simplified (because some conditions are simplified, and may become constant) and sequentialized again.

5 Refinements, First Proposals

When the analysis fails in showing the property, it can be due to overapproximation. Two kinds of approximation are used in linear relation analysis: in replacing set union by convex closure, and in extrapolating sequence limits by widening. Refinements are used to avoid convex closures. Convex closures are applied in two places:

- when merging transition results, at the level of an abstract state;
- when merging results of test branches, in sequentialized transitions.

Corresponding to these two cases, we distinguish between two kinds of refinements:

1. **Control structure refinement** consists is splitting some abstract states, thus obtaining a more accurate covering;
2. **Transition refinement** consists in opening some tests in sequentialized transitions, thus improving the precision of post- and pre-conditions.

Of course, we try to perform these refinements only when they appear to be necessary. We sketch below some ideas which are under experimentation.

5.1 Control structure refinement

Two cases strongly suggest that a state s should be split:

1. When the transition from s contains a test, the condition c of which only concerns state variables, and the branches of which lead to different states. In that case, it is surely interesting to split s into $s \wedge c$ and $s \wedge \neg c$.
2. Let s be an abstract state, we have (see §4.1.3)

   $$reach_s = \bigsqcup_{s'} post_{s',s} = \bigsqcup_{s'} \bigsqcup_{i} (post_{s'}^{(i)} \cap def_s)$$

   Let us call each term $(post_{s'}^{(i)} \cap def_s)$ a *contribution* to $reach_s$. Let $\mathcal{C}_s$ be the set of contributions to s:

   $$reach_s = \bigsqcup_{\varphi \in \mathcal{C}_s} \varphi$$

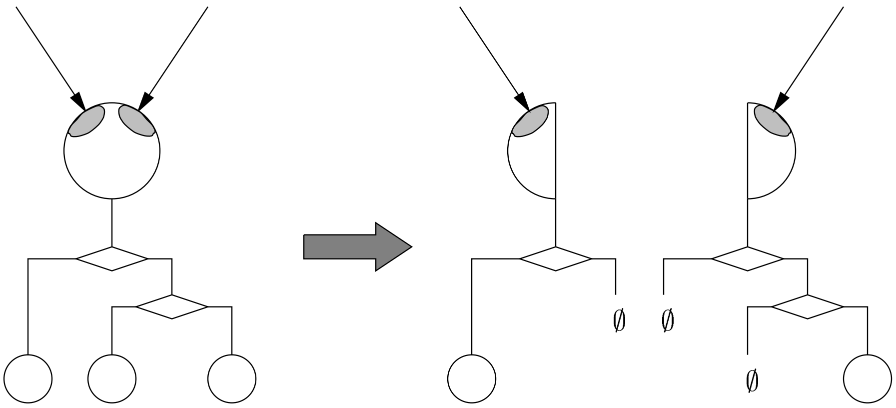


Figure 1: Splitting a state according to its contributions

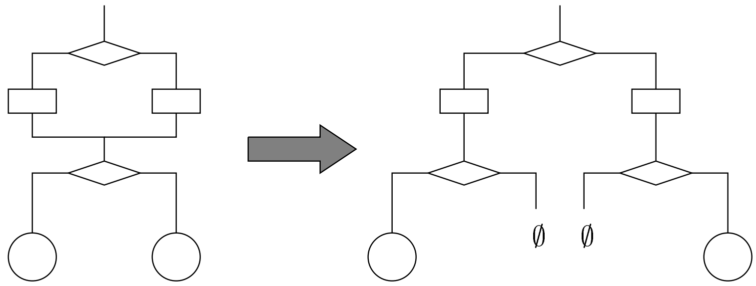


Figure 2: Refining a transition according to successor states

Now, for each contribution $\varphi \in \mathcal{C}_s$, let $post_s(\varphi)$ be its postcondition by the transition function from s. Obviously, a state s' is a successor of s only if $def_{s'}$ intersects $post_s(\varphi)$, for some $\varphi \in \mathcal{C}_s$. As a consequence,

$$\mathcal{S}_1 = \{s' \mid \exists \varphi \in \mathcal{C}_s, def_{s'} \cap post_s(\varphi) \neq \emptyset$$

is a better approximation of the set of successors of s than

$$\mathcal{S}_2 = \{s' \mid def_{s'} \cap post_s \neq \emptyset\}$$

If these two sets are different, it means that the convex closure performed in $reach_s$ loses information about the successor relation: states that belong to $\mathcal{S}_2 \setminus \mathcal{S}_1$ are wrongly considered to be successors of s. As a consequence, in this case, s should be split according to its contributions (see Fig. 1).

5.2 Transition refinement

In the same way, if it appears, in a sequentialized transition, that the branches of a closed test, considered separately, would lead to a strict subset of successor states, with respect to those which seem accessible in the standard case (convex closure of the results of the branches), the test should not be closed (see Fig. 2).

6 Further Work

At this stage, we have built a general workbench to make experiments and to adjust our heuristics and refinement strategy. Of course, the main part of the work remains to be done. The last year of the project will be devoted to dealing with examples of increasing complexity, which, hopefully, should allow us to make these choices.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 138:3–34, January 1995.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *ICALP'90*, 1990.

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

[Ray91] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : Le générateur de code Lustre-v3. Thesis, Institut National Polytechnique de Grenoble, November 1991.