A Lemma Generator Powered by Quantifier Elimination and Hull Computation.

Adrien Champion[†‡], Rémi Delmas[†], Michael Dierkes[‡]

[†] ONERA - The French Aerospace Lab - Toulouse - France
{adrien.champion,remi.delmas}@onera.fr
[‡] Rockwell Collins France
mdierkes@rockwellcollins.com

Abstract. This paper presents *hullQe*, a backward, property directed lemma generation algorithm for safety proof objectives on transition systems. It uses an SMT-based quantifier elimination algorithm to repeatedly compute the pre-image of a set of states violating a given proof objective through the transition relation. The algorithm can terminate if the pre-image intersects the initial states or if a fixed point is reached. Termination however is not our main concern. Rather, hullQe is designed as an invariant strengthening method to help other formal methods analyze problematic systems and proof objectives. Indeed, a key feature of the proposed approach is the simplification of the intermediate pre-image expressions through exact and inexact convex hull computation. This simplification allows to discover relational invariants over state variables of the system automatically, when other lemmata discovery methods such as abstract interpretation or template based methods require manual intervention. These lemmata can be communicated to other, cooperating formal methods in order to help them reach a conclusion faster, or at all; for instance by finding lemmata making the proof objectives 1-inductive instead of non-inductive or k-inductive, thus making the verification more scalable. The approach is illustrated through a simple example and through a piece of industrial code by Rockwell Collins. An implementation of the algorithm in a collaborative verification framework including k-induction and abstract interpretation is discussed.

1 Introduction

Quantifier Elimination (QE) procedures are very attractive for verification applications [8,21,19], but are also known for being very costly (double exponential in the number of variables in the worst case for linear real arithmetic for instance). However, in [16], Monniaux introduces a QE procedure based on lazy model enumeration using SMT-solvers [2] and polyhedral projection [15]. The practical efficiency of this QE technique makes it, in our experience, usable as a building block for symbolic reachability and lemma generation algorithms. In this paper, we propose one such algorithm, in which QE and additional processing are used to compute, given a transition system and a safety property to analyze, potential lemmata about the sate space of a transition system in a property directed way. As the algorithm progresses, it produces richer and richer

formulations of the original proof objective. The strength of the proposed algorithm is that it is able to infer relational invariants over the state variables of the system which have proved very valuable when used in cooperation with a k-induction engine [20] and an abstract interpretation engine [7], making some properties become *1-inductive* instead of *k-inductive* originally – where k depends on the system's parameters – or even not *k-inductive* at all for any k.

The proposed approach is particularly relevant in our context, critical embedded software verification for aerospace systems. Before being embedded, such pieces of code have to be specified, implemented and verified in accordance with relevant standards (*e.g.* the DO-178C[1]). Certification being a very expensive process, in order to set up a *proof based approach* to software certification (as opposed to a *test based approach*) one could imagine using a mature and trustworthy technique as central and qualified proof framework, such as k-induction[2], and using lemma generation techniques which are not certified, but whose results can be easily proved together with the original proof objective using the simpler and certified framework.

The paper is structured as follows: Section 2 presents the transition system formalism accepted by the algorithm, Monniaux's original QE algorithm, along with adaptations to linear integer logic and booleans. Section 3 introduces the core of the lemma generation algorithm, an iterated property directed pre-image computation using quantifier elimination. Sections 4 and 5 introduce the novelty of the proposed algorithm, exact and inexact *hullification*. Combining pre-image computation and hullification, we obtain the lemma generation algorithm hullQe (for **hull**ification and **QE**). hullQe's implementation is discussed in Section 7 along with its integration in our collaborative formal framework [6], while Section 6 discusses related work. Last, Section 8 concludes the paper and presents perspectives to our work.

2 Preliminaries

2.1 Transition Systems

Our work targets the formal verification of safety critical embedded software in the aerospace domain. These systems are usually specified using data-flow languages such as Lustre [12], SCADE[3] or Matlab Simulink[4]. We hence adopt a notion of transition system suitable to capture the semantics of these specification languages, and simple enough to ease the design and implementation of verification algorithms. So, we consider transition systems $\Sigma = \{v, D, I, T\}$ where[5]:

[1] http://www.rtca.org/onlinecart/product.cfm?id=501

[2] *c.f.* Prover Certifier in railway applications: http://www.prover.com/products/prover_certifier/

[3] http://www.esterel-technologies.com/products/scade-suite/

[4] http://www.mathworks.com/products/simulink/

[5] Please note that we will distinguish between $=$ and $\equiv$; $=$ will be used for definitions (see Equation 2 for instance), and $\equiv$ for logical equivalence (*e.g.* $p \wedge \neg p \equiv \mathbf{false}$).

- $v = (v_1, \ldots, v_n)$ is a vector of state variables ranging over the domain $D = \mathbb{B}^i \times \mathbb{Z}^j \times \mathbb{Q}^k$. A valuation s of the state vector is called a *state* of the system;
- $I(v) : D \to \mathbb{B}$ is the initial state predicate, such that $\forall s \in D$, $I(s) \equiv \textbf{true}$ if and only if s is an initial state, $I(s) \equiv \textbf{false}$ otherwise;
- $T(v, v') : D^2 \to \mathbb{B}$ is the transition relation of the system, in which v represents the current state, and v' represents the next state, and such that $\forall (s, s') \in D^2$, $T(s, s') \equiv \textbf{true}$ if and only if s' is a successor of s.

I and T are expressed in a quantifier-free logic combining propositional logic (Prop), linear real arithmetic (LRA) and linear integer arithmetic (LIA) (without coercions from integer to real or conversely).

A sequence of states $(s_0, \ldots, s_n)$ is called a *trace* of the system if $T(s_i, s_{i+1})$ evaluates to **true** for each $i \in [0, n-1]$. It is called an *initialized trace* if $I(s_0)$ evaluates to **true**. A state is called *reachable* if there exists an initialized trace containing the state. We call $R \subseteq D$ the *set of reachable states* of the system.

A *proof objective* over Σ is specified by a predicate P expressed over v. The verification problem consists in determining whether or not there exist reachable states of Σ such that P evaluates to **false**.

In the rest of the paper we will consider states which do not violate P, but from which a state violating P can be reached in a finite number of transitions. We will refer to these states as *gray states*, and to their set as the *gray state space* G, which is of course a subset of the state space D. If the proof objective P holds, then obviously the gray state space is not reachable from I.

2.2 Quantifier Elimination

The algorithm proposed in this paper relies heavily on Quantifier Elimination (QE). QE yields, for a quantifier-free formula F and some vector of variables $\boldsymbol{v} \subseteq FV(F)$ (where $FV(F)$ is the set of variables appearing in F), a formula $\mathcal{G}$ such that $\mathcal{G}$ is quantifier-free and logically equivalent to the original quantified formula, $(\exists \boldsymbol{v}, F) \equiv \mathcal{G}$, and $\boldsymbol{v} \cap FV(\mathcal{G}) = \emptyset$. We will write $\texttt{QE}(\boldsymbol{v})(F) = \mathcal{G}$.

Even if various approaches to QE exist, such as LDD:s [5] or LinAIG:s [19], we chose to use Monniaux's algorithm [16], based on SMT and polyhedral projection. This choice was motivated by our familiarity with SMT solvers and the simplicity of the algorithm. It can be adapted to handle not only reals but also integers and booleans as described in Section 7.1, and is easy to tailor to fit our needs. Another convenient aspect is that its results are produced as formulae in DNF, easing the *hullification* process (see 4). Yet, the hullQe technique itself does not depend on any quantifier elimination method in particular.

3 Extracting Property Directed Information From A Transition System

3.1 Backward Reachability by Quantifier Elimination

In this section we describe the core of the hullQe algorithm, *i.e.* an iterated property-directed pre-image computation on transition systems, and illustrate it on a small example.

Algorithm 1 Core pre-image computation of the hullQe algorithm.

```
G ← QE(e)(P(v) ∧ T(v, v') ∧ ¬P(v'))
H ← G
over ← false
while (¬over) do
   G ← QE(e)(P(v) ∧ T(v, v') ∧ makeNext(G))
   if (G ≡ false) then
      over ← true
   else
      if (SMTsolver.checkSat(H ∧ ¬G) == UnSat) then
         over ← true
      else
         H ← H ∨ G
      end if
   end if
end while
return H
```

Given a transition system $\Sigma = \{v, D, I, T\}$ and a proof objective P, let us note v_P the subset of v actually appearing in the cone of influence of $P(v)$. The set of variables we want to eliminate is $e = (v \setminus v_P) \cup v'$. It contains the current-state variables **not** appearing in $P(v)$, and all the next-state variables. Let us consider the following formula:

$$\exists e,\ P(v) \wedge T(v, v') \wedge \neg P(v'). \tag{1}$$

which characterizes, in intention, the set of states satisfying P from which a state violating P can be reached in a single transition. By using `QE`, to eliminate e:

$$\mathcal{G}_1(v) = \texttt{QE}(e)(P(v) \wedge T(v, v') \wedge \neg P(v')) \tag{2}$$

we obtain a formula $\mathcal{G}_1(v)$ characterizing exactly the same states as (1), but in extensional form, and only in terms of v_P variables. We can then proceed by using `QE` on the following formula (where $\mathcal{G}_i(v')$ is obtained by substituting occurrences of v variables by their v' counterparts):

$$\mathcal{G}_2(v) = \texttt{QE}(e)(P(v) \wedge T(v, v') \wedge \mathcal{G}_1(v')) \tag{3}$$

which yields $\mathcal{G}_2(v)$, a formula characterizing states for which P holds but from which a state violating it can be reached in two transitions. Obviously, the idea is to iterate to make our under-approximation of the gray state space more and more precise:

$$\mathcal{G}_{k+1}(v) = \texttt{QE}(e)(P(v) \wedge T(v, v') \wedge \mathcal{G}_k(v')). \tag{4}$$

The exact characterization of the gray states leading to a violation of P in k transitions or less is then $\mathcal{H}_k(v) = \bigvee_{1 \leq i \leq k} \mathcal{G}_i(v)$ (due to the particular QE method used, it is a formula in DNF).

3.2 A First, Simple Algorithm

A pseudo-code description of the pre-image computation algorithm is given in Algorithm 1. The `makeNext` function syntactically substitutes the current state variables with primed variables representing the next state, and allows to iterate the pre-image computation, until no new states are discovered.

```
 1: node top(a,b,c: bool) returns (o1, o2, ok: bool);
 2: var
 3:   x, y, pre_x, pre_y: int;
 4:   n1, n2: int;
 5: let
 6:   n1    = 10;
 7:   n2    = 6;
 8:   x     = if (b or c) then 0 else (if (a and pre_x < n1) then pre_x + 1 else pre_x);
 9:   y     = if   (c)    then 0 else (if (a and pre_y < n2) then pre_y + 1 else pre_y);
10:   o1    = x = n1;
11:   o2    = y = n2;
12:   ok    = o1 => o2;
13:   pre_x = 0 -> pre(x);
14:   pre_y = 0 -> pre(y);
15:   prove(ok);                  (* main proof objective *)
16:   prove(0 <= x and x <= 10); (* range lemma *)
17:   prove(0 <= y and y <=  6); (* range lemma *)
18: tel
```

Fig. 1: A Lustre program using two counters.

A slight performance improvement is obtained by adding the constraint $v \neq v'$ to the transition formula. It avoids the re-discovery of states s such that $T(s,s)$ and limits the size of the pre-image (it is not the only benefit of this extra constraint, as will be discussed in Section 4.2). From now on we will consider that the constraint $v \neq v'$ is automatically added to the transition relation.

3.3 A Simple Example

As an example, let us consider the Lustre program shown in Figure 1. This example is rather representative of functions mixing discrete and numeric logic used for discrete input filtering in embedded systems. The program uses integers and booleans, which are not handled by Monniaux's original QE algorithm. See Section 7.1 for adaptations of the QE algorithm allowing to handle these systems. Its transition relation can be modeled as follows:

$$\begin{aligned} T(x,y,x',y',a,b,c) = \quad & \Big(\Big((b \vee c) \wedge (x=0)\Big) \vee \Big(a \wedge \neg b \wedge \neg c \wedge (x<10) \wedge (x' = x+1)\Big) \\ & \qquad \vee \Big(\neg b \wedge \neg c \wedge (\neg a \vee \neg(x<10)) \wedge (x'=x)\Big)\Big) \\ & \wedge \Big(\Big(c \wedge (y=0)\Big) \quad \vee \Big(a \wedge \neg c \wedge (y<6) \wedge (y' = y+1)\Big) \\ & \qquad \vee \Big(\neg c \wedge (\neg a \vee \neg(y<6)) \wedge (y'=y)\Big)\Big). \\ & \wedge (x \neq x' \vee y \neq y') \quad \text{(reflexivity breaker)} \end{aligned}$$

The proof objectives (lines 15-17) are formalized as follows:

$$P(x,y) = (0 \leq x \leq 10) \wedge (0 \leq y \leq 6) \wedge (x = 10 \rightarrow y = 6) \tag{5}$$

The intermediate pre-image results ($\mathcal{G}_i$-s in the algorithm) are first $x = 9 \wedge 0 \leq y < 5$, then $(x = 8 \wedge 0 \leq y < 4)$, $(x = 7 \wedge 0 \leq y < 3) \ldots$ A fixed point is quickly reached:

$$\begin{aligned} \mathcal{H}_5 \equiv (x = 9 \wedge 0 \leq y < 5) \vee (x = 8 \wedge 0 \leq y < 4) \vee (x = 7 \wedge 0 \leq y < 3) \\ \vee (x = 6 \wedge 0 \leq y < 2) \vee (x = 5 \wedge 0 \leq y < 1) \end{aligned} \tag{6}$$

This result, however interesting (the initial states do not intersect the gray states, which entails that P holds), still leaves something to be desired. It just consists in an enumeration of the gray states without any relational generalization whatsoever, which would not be feasible for large systems. A more interesting information about the system would be $8 \leq x \leq 9 \wedge 0 \leq y < x - 4$ for $\mathcal{G}_2$ for instance. Also, and more importantly, we do not really want hullQe to prove anything by itself because of our context (cf. Section 1): we would rather find strengthening lemmata allowing the k-induction engine to conclude easily. This relational generalization will be obtained thanks to *exact hullification*, described in Section 4, and thanks to *inexact hullification*, described in Section 5.

4 Convex Hulls

In this section, we describe how $\mathcal{H}$, the intermediate disjunction of polyhedra produced by the pre-image algorithm on line 13 of Algorithm 1, is simplified by searching for an equivalent but smaller formula through exact convex hulls computation. First of all, let us assume two functions: `convexHull` which computes a convex hull of two polyhedra given as parameters, and `convexHullExact` performing the same computation as `convexHull` but failing if the resulting convex hull is not exact (*i.e.* contains strictly more points than the union of its two arguments).

4.1 Exact Convex Hull Computation

Given a list of polyhedra, the algorithm starts from a polyhedron p_0 and tries to compute the **exact** convex hull with every other polyhedron separately. When an exact convex hull is found with polyhedron p_i, the algorithm continues after replacing p_0 by the hull and discarding p_i. If the hull with polyhedron p_i is not exact, p_i is put aside to be examined again later. If at least one exact convex hull was found once there are no more polyhedra to check, the algorithm starts over with the previously computed hull as p_0 and the polyhedra *previously put aside* until a fixed point is reached. The algorithm then carries on with the remaining polyhedra, and in the end returns the list of exact convex hulls it found and polyhedra for which there was none. The process is iterated again on the new list of polyhedra (because some separately computed hulls could have become *mergeable*) until a fixed point is reached. Upon termination, the polyhedra returned are such that (i) their union is equisatisfiable to the union of the input polyhedra and (ii) none of them can be merged exactly with another one.

However, it can be the case that the only way to find an exact convex hull between three or more polyhedra is to merge them at the same time; in this case, the algorithm described above would not find it. It is worth noting that the returned list of convex hulls is still an exact characterization of the gray states encountered so far, as all inexact merges are rejected. In the rest of this paper, we will call *exact hullification* this convex hull computation, as opposed to the *inexact hullification* introduced later in Section 5.

Algorithm 2 Core hullQe algorithm, with convex hull computation.
(Here we assume an implicit conversion of polyhedra disjunctions from their formula representation to a list representation suitable for the `hullify` function and back.)

```
1: T ← T ∧ v ≠ v'
2: G ← hullify(QE(e)(P(v) ∧ T(v, v') ∧ ¬P(v')))
3: H ← G
4: over ← false
5: while (¬over) do
6:     G ← hullify(QE(e)(P(v) ∧ T(v, v') ∧ makeNext(G)))
7:     if (G ≡ false) then
8:         over ← true
9:     else
10:        if (SMTsolver.checkSat(H ∧ ¬G) == UnSat) then
11:            over ← true
12:        else
13:            H ← hullify(H ∨ G)
14:        end if
15:    end if
16: end while
17: return H
```

4.2 Using Convex Hull Computation and Reflexivity Breaking

Convex hull computation is introduced in the core hullQe algorithm both at the QE and at the hullQe level, as described in Algorithm 2.

One might be surprised to find two convex hull computations, one on line 6 (called $\mathcal{G}_{E,i}$) and another on line 13 (called $\mathcal{H}_{E,i}$). They are computed separately because each of them serves a different purpose. On the one hand, $\mathcal{G}_{E,i}$ is used to compute the next fringe $\mathcal{G}_{i+1}$. We do not want to iterate the pre-image computation on $\mathcal{G}_i$, since the cost of QE algorithm greatly increases with the number of boolean atoms of the input formula, and $\mathcal{G}_{E,i}$ has less atoms than $\mathcal{H}_{E,i}$. We could use $\mathcal{H}_{E,i}$ for pre-image computation, but this formula is in general (a lot) larger, and would produce a result re-characterizing all the states found so far (as $\mathcal{H}_{E,i}$ characterizes the gray states leading to $\neg P$ in i transitions **or less**). Also, calling hullification on such a result would recompute all the convex hulls found up to this point plus those for the new polyhedra if any.
On the other hand, $\mathcal{H}_{E,i}$ is the formula containing the most information. It characterizes all the gray states found so far wrapped in convex hulls when possible. It is also used for the fixed point check (line 10), and optionally to check whether the original proof objective holds by verifying if some of the initial states are gray.

The *reflexivity breaking* constraint, introduced in Section 3.2, has a positive effect on the convex hull computation. We observed that:

- it reduces the time spent at every QE iteration by reducing the size of the input formula;
- it saves a great deal of convex hull (re)computation, by making the convex hulls calculus incremental (without it, every hull computation performed would be re-done at every following step since the QE call would output all the gray states found so far again with *several* new ones);

4.3 Simple Example (continued)

Going back to the example of Figure 1, the formula characterizing the fixed point obtained using the exact hullification is $5 \leq x \leq 9 \wedge 0 \leq y < x - 4$. Also,

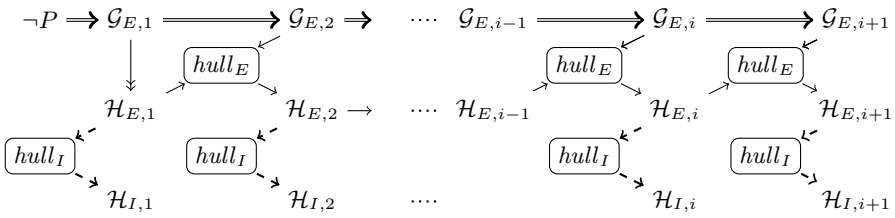


Fig. 2: Flow chart of pre-image, exact and inexact hulls computations.

as expected, the intermediary results (the $\mathcal{H}_i$-s) become $x = 9 \wedge 0 \leq y < 5$ for $i = 1$, $8 \leq x \leq 9 \wedge 0 \leq y < x - 4$ for $i = 2$, etc. Since these equations characterize the gray state space, their negation must be used as when trying to strengthen the proof objective under investigation using a forward method. Let us consider in particular $y < x - 4$, appearing on hullQe's second iteration. Its negation, $y \geq x - 4$ is a strengthening lemma which makes the original proof objective 1-inductive. In its original form, the proof objective is k-inductive for k proportional to the difference of x and y's upper bounds, under the loop free path assumption. In practice, k-induction alone will not be able to prove the property if that k is too large (timers of a few hundred steps are not uncommon in reactive systems). The range lemmas on x and y can be found easily using abstract interpretation for any values of the upper bounds. Yet, proving the original proof objective using AI alone requires non trivial domain partitioning directives. Using the AI's range lemmata, hullQe discovers a strengthening lemma in two iterations regardless of the model parameters. High level cooperation strategies between these three techniques will be discussed in Section 7.

5 Inexact Convex Hulls

Even though hullQe with exact hullification can already provide valuable information about a system, we would like to have an abstraction mechanism for two different, yet not unrelated, purposes. First, to perform abstraction on the disjuncts of a pre-image in order to over-approximate the gray state space: the goal is to offer an alternative, inexact pre-image using heuristics to try to reach useful information – such as a lemma making the proof objective 1-inductive, or to offer efficient partitioning directives for Abstract Interpretation. In this case, abstraction is not performed inside the hullQe engine, but on the information communicated outside hullQe. The second approach is to perform abstraction while iterating the pre-image computation in order to achieve better scalability. This section only deals with the former approach, while the latter is currently under investigation. The abstraction mechanism we set up is called *inexact hullification*, as opposed to the exact hullification developed in Section 4. This technique is a heuristic; the algorithm is the same as exact hullification but polyhedra are merged (inexactly) provided they satisfy a given criterion (as opposed to being merged only if the result is exact in exact hullification).

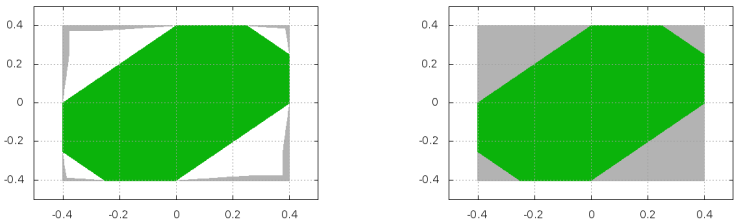
5.1 Inexact Hullification

The criterion retained to inexactly merge two hulls is that they must have at least one point in common. The test itself consists in a simple satisfiability

```
node top(input1, input2, input3: real) returns (output: real);
var
  equalized1, equalized2, equalized3: real;
  equalization1, equalization2, equalization3 : real;
  satCentering, centering : real;
  df1, df2, df3, st1, st2, st3, c1, c2, c3, d1, d2, d3 : bool;
  check: bool;
let
  assert (input1 <  0.2); assert (input1 > -0.2);
  assert (input2 <  0.2); assert (input2 > -0.2);
  assert (input3 <  0.2); assert (input3 > -0.2);

  equalized1 = input1 - equalization1;
  df1 = equalized1 - output;
  st1 = if (df1 > 0.5) then 0.5 else (if (df1 < -0.5) then -0.5 else df1);
  equalization1 = 0.0 -> pre (equalization1) + (pre (st1) - pre (satCentering)) * 0.05;

  equalized2 = input2 - equalization2;
  df2 = equalized2 - output;
  st2 = if (df2 > 0.5) then 0.5 else (if (df2 < -0.5) then -0.5 else df2);
  equalization2 = 0.0 -> pre (equalization2) + (pre (st2) - pre (satCentering)) * 0.05;

  equalized3 = input3 - equalization3;
  df3 = equalized3 - output;
  st3 = if (df3 > 0.5) then 0.5 else (if (df3 < -0.5) then -0.5 else df3);
  equalization3 = 0.0 -> pre (equalization3) + (pre (st3) - pre (satCentering)) * 0.05;

  c1 = equalized1 > equalized2; c2 = equalized2 > equalized3; c3 = equalized3 > equalized1;
  output = if (c1 = c2) then equalized2 else (if (c2 = c3) then equalized3 else equalized1);
  d1 = equalization1 > equalization2;
  d2 = equalization2 > equalization3;
  d3 = equalization3 > equalization1;

  centering = if       (d1 = d2) then equalization2
              else (if (d2 = d3) then equalization3
              else                    equalization1);
  satCentering = if       (centering >  0.25) then  0.25
                 else (if (centering < -0.25) then -0.25
                 else                              centering);
  check = ( equalization1 <=  2.0 * 0.2) and ( equalization1 >= -2.0 * 0.2) and
          ( equalization2 <=  2.0 * 0.2) and ( equalization2 >= -2.0 * 0.2) and
          ( equalization3 <=  2.0 * 0.2) and ( equalization3 >= -2.0 * 0.2);

  prove(check);
tel
```

Fig. 3: Rockwell Collins' Triplex Voter.

check using an SMT solver after asserting both polyhedra. Our polyhedra are not necessarily closed, so checking them for intersection as they are entails that some polyhedra would not be merged together despite having an adjacent edge, such as $x >= 0 \wedge y >= 0 \wedge x + y < 1$ and $x <= 1 \wedge y <= x \wedge x + y >= 1$. In our experience it is better to check for intersection on closed version of the polyhedra, and then merge the originals together if the test succeeds. At each step, we perform inexact hullification on $\mathcal{H}_{i+1}$ and then send both the exact and the inexact version to the rest of the framework. Figure 2 describes the hullification process in a hullQe run: Subscripts E and I are used to distinguish between exact and inexact hullification operations and results. For example, $\mathcal{H}_{E,2}$ is the result of the exact hullification $hull_E$ between $\mathcal{H}_{E,1}$ and $\mathcal{G}_{E,2}$.



(a) First pre-image and strengthening lemma found by hand

(b) Inexact hullification

Fig. 4: hullQe on the duplex voter

5.2 Application to Rockwell Collins' Triplex Voter

Using inexact hullification, we managed to prove the numerical stability of Rockwell Collins' triplex voter [10] (without fault detection, code in Figure 3). In [10], the strengthening lemma was found using a template based approach, which requires an appropriate choice of the templates, whereas hullQe does not need any user interaction. The following strengthening lemma was extracted from hullQe's first pre-image and makes the original proof objective 1-inductive:

$$-0.9 \leq equalization1 + equalization2 + equalization3 \leq 0.9 \tag{7}$$

It is weaker than the lemma found in [10]:

$$-2/3 \leq equalization1 + equalization2 + equalization3 \leq 2/3 \tag{8}$$

because it is built by pre-image computation starting from the negation of the proof objective, *i.e.* outside of the reachable state space provided the property holds and gradually getting closer to it, while lemma (8) is built from the initial states using k-induction's counterexamples to widen an under approximation of the reachable state space. Also, it is worth noting the quantitative difference between the exact $\mathcal{H}$ and the inexact one. On the triplex voter, the exact version contains about thirty disjuncts (*i.e. distinct ways* of violating the property in one transition), whereas the inexact $\mathcal{H}$ has four disjuncts.

A geometric illustration of the inexact hullification process is given for a simplification of Rockwell Collins' triplex voter on Figure 4 to a two inputs system instead of three, dubbed the *duplex voter* – created for experimental purposes. The central octagon corresponds to the strengthening lemmata from [10]. The gray triangles represent hullQe's $\mathcal{H}$ on the first iteration before (Figure 4a) and after (Figure 4b) inexact hullification. Additional processing, developed in the next section, is needed to extract strengthening lemmata from the gray state space characterization.

$$
\begin{aligned}
\mathit{init} &= (x = 0 \wedge y = 0) \\
c_1 &= (x = 10 \wedge 0 \leq y < 6) \\
c_2 &= (x = 10 \wedge 0 \leq y < 5) \\
&\vdots \\
c_6 &= (x = 10 \wedge 0 \leq y < 1)
\end{aligned}
$$

R_0	R_1	R_2	$\ldots$	R_7	R_8
init	$\neg c_1$	$-$	$\ldots$	$-$	$-$
init	$\neg c_1 \wedge \neg c_2$	$\neg c_1$	$\ldots$	$-$	$-$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
init	$\neg c_1 \wedge \cdots \wedge \neg c_6$	$\neg c_1 \wedge \cdots \wedge \neg c_5$	$\ldots$	$\neg c_1$	$-$
init	$\neg c_1 \wedge \cdots \wedge \neg c_6$	$\neg c_1 \wedge \cdots \wedge \neg c_6$	$\ldots$	$\neg c_1 \wedge \neg c_2$	$\neg c_2$

Fig. 5: Application of PDR on the double counter

6 Related Work

A very similar approach to hullQe was developed in [8] by de Moura *et al.* for counterexample refutation, but it differs in the sense that its purpose is to generalize step counter-examples in a k-induction engine to the states leading to them in k transition(s), and not potential invariant generation in a collaborative framework (not to mention exact and inexact hullification). Nevertheless, combining hullQe with k-induction (and abstract interpretation) yields interesting results as we saw, and as will be discussed in 7. Also worth mentioning is the work of Jeannet [13] who investigated dynamic partitioning in a backward abstract interpretation approach. While Jeannet's main concern was to find a way to automatically partition the abstract domains, ours is to some extent a dual problem, as pre-image computation using QE yields a completely partitioned expression of the state space, and raises the need for automatic simplification *a posteriori*, as discussed in Section 4.

Since both of them are property directed reachability methods, an attempt to relate Bradley's algorithm PDR [3,11] to hullQe appears legitimate. hullQe being designed for numerical system processing, and PDR being – in its current form – purely propositional, a direct comparison is not possible. Nevertheless, the run of a naïve adaptation of PDR on the double counter shown in Figure 5 allows us to highlight a few connections between the two. The run generates the set of gray states computed in Section 3.3 by hullQe without hullification: PDR lazily enumerates gray states and blocks their generalizations one by one until termination to either obtain a strengthening lemma or a counterexample; on the other hand, hullQe takes a more greedy approach and computes the whole pre-image using QE before broadcasting potential invariants found by hullification at each step. Intuitively, hullQe's iterations are more expensive, but unlike PDR, strengthening lemmata can be found at any time without exhaustively enumerating the gray states – *e.g.* on the second iteration for the double counter and on the first iteration for the triplex voter. Last, we believe that despite their

differences, some of PDR's ideas can be transposed to hullQe, such as the frame structure. More perspectives are discussed in Section 8.

7 QE Adaptation, Implementation and Communication

7.1 Adaptation of QE and Hullification to Discrete Systems

In its original version, Monniaux's QE algorithm only handles linear real arithmetic. In order to perform quantifier elimination on formulae of the logic used to specify transition systems introduced in the previous section, we adopt the following encoding: boolean variables are modeled by introducing real variables and real inequalities (*i.e.* a boolean variable v becomes $0.0 \leq v_{alt}$, and $\neg v$ becomes $0.0 > v_{alt}$, where v_{alt} is a fresh variable introduced specially for v), and integers are handled by relaxation to reals. However, polyhedra libraries, such as the Parma Polyhedra Library which is used in our implementation, model polyhedra using real (rational) values. Therefore, PPL can output non-empty polyhedra when considering all the variables as reals, but empty when some of them are actually integers. To cope with this problem, a satisfiability check is performed *a posteriori* on each polyhedron to make sure it contains at least one solution consistent with the effective sorts of the variables.

For the same reasons, the `convexHullExact` primitive offered by PPL can fail on the real relaxation, even if the hull is exact when interpreted over the actual sorts of the variables. In this case, instead of computing the exact hulls of the polyhedra using `convexHullExact`, we use `convexHull` and check afterwards for equisatisfiability between the inexact hull and the disjunction of the two original polyhedra, using the effective sorts of the variables. This solution is not satisfactory, and better ways to handle discrete variables will be studied.

7.2 Implementation, Communication

The authors implemented the algorithm presented on Figure 2 using Scala [17], Microsoft Research's Z3 SMT solver [9] and the Parma Polyhedra Library [1] for projections. Our collaborative framework [6] provides three analysis methods: abstract interpretation [18], k-induction and hullQe. Analyzes are conducted as follows. First, abstract interpretation with intervals as abstract domains is used in an attempt to infer bounds on the state variables. hullQe and k-induction are then run in parallel (using whatever information the AI discovered): hullQe sends potential lemmata to the k-induction engine which tries to prove the main proof objective in conjunction with the lemmata in an incremental fashion – *i.e.* any lemma the k-induction falsifies is removed and the proof attempt carries on with whatever information has not been invalidated yet. Once a sufficient lemmata set has been found (*i.e.* the k-induction proves the main proof objectives), it is minimized and the proof is confirmed using Tinelli's KIND k-induction tool [14]. The minimization consists in removing elements of the lemmata set one by one, checking if the proof still holds, and putting them back if it does not. This minimization is being changed in favor of a more efficient method based on unsat cores. The whole strategy detailed above is performed automatically without

external intervention.
The actor oriented nature of our framework allows to easily run the different techniques in parallel. Additionally, it is possible to instantiate several different k-induction actors to make them run on different sets of lemmata. In our case, one for the negation of the inexact $\mathcal{H}_i$, and one for the negated atoms of both the exact and inexact $\mathcal{H}_i$. Hence, we take advantage of the incremental nature of our k-induction engine: since $\mathcal{H}_i$ is in DNF, negating it and propagating the negations results in a CNF formula, whose conjuncts can be streamed to the k-induction as soon as they are discovered by hullQe. The same goes for the negated atoms which are sent as different proof objectives.
This entails that we do not have to fear to fail the proof for sending falsifiable potential lemmata, since they will be discarded and the analysis will continue. The only problem would be choking the k-induction with too many of them, but in our experience the number of lemmata the hullQe communicates stays reasonable. For instance, hullQe only communicates about 20 lemmata on the first iteration of the duplex's analysis (8 of which are actually needed by k-induction to conclude). For the triplex voter, this number does not exceed 150. Regarding performance, the double counter example is solved in less than three seconds, the duplex in about one minute, and the triplex in a little more than two minutes.

8 Conclusion And Future Work

In this paper, we have presented a new property directed lemmata generator for transition systems, which is based on a generic quantifier elimination algorithm. The most notable feature of the proposed analysis is the simplification of intermediate pre-image results using convex hulls computations, which can allow to discover relational invariants of the transition systems in a property directed way. This feature becomes valuable in a cooperation framework, where different techniques such as k-induction and abstract interpretation can be used to analyze a common proof objective and mutually enhance their results to conclude a proof that no technique alone can conclude easily or at all. The work presented here is still in an early state and perspectives are many.

First, at the QE level, we think performance gains can be obtained by processing boolean variables using a dedicated boolean QE algorithm instead of encoding them using real/integer variables. Also, it seems natural to try to perform abstraction on the pre-images hullQe iterates on, for example using the mechanism mentioned at the beginning of Section 5, to accelerate hullQe's progression. Even if we did not witness any blowup on our examples in the amount of potential lemmata sent to the k-induction, more efficient ways to identify relevant information still need to be found and enforced. Regarding abstract interpretation, we believe information from hullQe could be used to infer partitioning, variable packing and/or domain selection automatically. Such tuning is usually performed by an expert user, to improve the precision of the analysis. Last, even if hullQe can fail to discover a strengthening lemma, the shape of relational potential lemmata can be transmitted to a template-based approach for parameter tuning. Consider that for some system hullQe produces a formula $a.x + b.y + c.z < d$ where x, y, z are state variables, but too weak to be a

strengthening lemma. A template based approach could try to find a value d' such that $a.x + b.y + c.z < d'$ is a strengthening lemma, in the spirit of Bradley's early work [4].

A detailed reflexion on all of these matters will be carried out to make the most out of hullQe features in a cooperative setting.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2), 2008.
2. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*. 2009.
3. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, 2011.
4. A. R. Bradley and Z. Manna. Verification constraint problems with strengthening. In *ICTAC*, 2006.
5. S. Chaki, A. Gurfinkel, and O. Strichman. Decision diagrams for linear arithmetic. In *FMCAD*, 2009.
6. A. Champion, R. Delmas, P.L. Garoche, and P. Roux. Towards cooperation of formal methods for the analysis of critical control systems. In *SAE Aerotech, to be published*, 2011.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
8. L. M. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a). In *CAV*, 2003.
9. L. M.ça de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
10. M. Dierkes. Formal analysis of a triplex sensor voter in an industrial context. In G. Salaün and B. Schätz, editors, *Proceedings of the 16th edition of FMICS*, volume 6959 of *LNCS*. Springer, 2011.
11. N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of IWLS*. IEEE/ACM, 2011.
12. N. Halbwachs. A synchronous language at work: the story of Lustre. In *Third ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'2005*, Verona, Italy, jul 2005.
13. B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1), 2003.
14. T. Kahsai and C. Tinelli. PKind: A parallel k-induction based model checker. In *PDMC*, 2011.
15. B. L. Kaluzny. Polyhedral computation: A survey of projection methods, 2002.
16. D. Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *LPAR*, 2008.
17. M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
18. P. Roux, R. Delmas, and P.L. Garoche. Smt-ai: an abstract interpreter as oracle for k-induction. *Electr. Notes Theor. Comput. Sci.*, 267(2), 2010.
19. C. Scholl, S. Disch, F. Pigorsch, and S. Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In *TACAS*, 2009.
20. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, 2000.
21. T. Sturm and A. Tiwari. Verification and synthesis using real quantifier elimination. In *ISSAC*, 2011.