1 Implementation of PDR/IC3 for Linear Integer Arithmetic.

- Akshay Naik, Devendra Reddy

1.1 Description of the PDR Algorithm

1.1.1 Overall Procedure

```
> PDR(I, T, P):
>   F = []
>   F_0 = I
>   n = 1
>   Q = []
>   if SAT(F_0 && !P):
>     Terminate "P not satisfied in Init."
>   F_1 = P //Since blocking will give this anyway.
>   while True:
>     if UNSAT(F_n && !P):
>       Propagate(n)
>       n++
>     else:
>       for cube in (F_n && !P)
>         Block(<s,n>)
```

1.1.2 Propagating Phase

```
> Propagate(n):
>   if frontier_index <= n+1:
>     F_(n+1) = True
>   for k in [1 .. n-1]:
>     for c in clauses(F_k) and not in clauses(F_(k+1)):
>       if UNSAT(F_k && T && !c'): //(F_k && T => c')
>         F_(k+1) := F_(k+1) && c
>         //Optional: Subsumption check to remove weaker clauses.
>         for d in clauses(F_(k+1)):
>           if UNSAT(c && !d):  //(c => d)
>             F_(k+1).remove(d)
>     if clauses(F_(k+1)) = clauses(F_k):
>       Terminate "P is valid in system."
>   return
```

1.1.3 Blocking Phase

If SAT($F_n \wedge\neg$ c $\wedge$ T $\wedge$ c'), then $\exists$ a cube $s \in F_n$, corresponding to the proof obligation `<s,n>`. All such cubes, s need to be blocked by calling the `Block<s,n>`. But by generalzing s inside `Block()` we hope to reduce computation on subsequent cubes.

Q is a priority queue(prioritized over i) consisting of elements $< c, i >$ where c is a cube and i is a frame number. $< c, i > \in Q$ implies that there exists a path of length $n - i$ from a state in c to $\neg P$. Here, F_n is the frontier frame; i.e. not necessarily true that $F_n \wedge \neg P$ is unsat.

Note: `Pre-image of s in F_i` = $\exists x'(F_i \wedge \neg s \wedge T \wedge \neg s')$

```
> Block(<s,n>): //<cube, frame num>
>   Q.add(<s,n>)
>   while !Q.empty():
>     <s,k> := Q.pop()
>     if k = 0:
>       Terminate "P not satisfied."
>     if UNSAT(F_k && s):
>     //(F_k => !s), s is blocked at k.
>       continue //look at next obligation.
>     if SAT(F_(k-1) && !s && T && s'):
>     //s isn't inductive
>     // s has predecessors in F_(k-1)
>       Pre := Pre-image of s in F_(k-1)
>       for cube in Pre:
>         Q.add(<cube,k-1>)
>       Q.add(<s,k>)
>     else:
>       c = generalizeUNSAT(s) //s=>c, but query still unsat.
>       //Strengthen
>       for i in [k .. 1]:
>         if !c in clauses(F_k): //syntactic check.
>           break
>         F_i := F_i && !c
>   return
```

1.1.3.1 Generalizing blocking clause from UNSAT query

When `UNSAT`$(F_{k-1} \wedge \neg s \wedge T \wedge s')$, instead of blocking by adding $\neg s$ to the frame, we need to find a stronger fact to block. That is, we need a cube, c such that $s \subseteq c$ or $s \Rightarrow c$ and `UNSAT`$(F_{k-1} \wedge \neg c \wedge T \wedge c')$.

As a starting point we get get the unsat core of the original query and use it to construct the clause c, by only keeping those constraints in s which appear in the unsat core. Further, we need to check that the generalized clause c does not intersect the initial states.

1.1.3.2 Generalizing cube in from SAT query

Given a pre-image as a set of cubes, we need a generalized cube c such that $c \Rightarrow$`Pre-image`, but `UNSAT(`$c \land \neg$`Init)`.

1.2 Implementation

We implemented algorithm in **Python**, using **Z3** for SMT queries.

Each frame is stored as a **ConjFml** object which is an extension of the Z3's **Goal** class. A **ConjFml** can be thought of as a list of formulas to be interpreted as a conjunction over all its elements.

Since we restrict each formula in a **ConjFml** object to be a clause then a frame can be represented as a **ConjFml**. This allows for easy and efficient checking of syntactic clause containment and clause propagation. Furthermore, every cube can also be represented as a ConjFml object.

Also, a goal object can be easily converted into a single formula which is a conjuction of the entire set of formulas. This is ideal for formulating SMT queries involving frames and cubes.

Performing existential quantification for pre-image computation returns a disjunction of goals, which can be converted into DNF using Z3's built-in tactics and tactic combinators. But we chose to write our own method as Z3 tactics required the input to be in CNF, and obtaining the pre-image in CNF using Z3's 'tseitin-cnf' tactic resulted in large number of cubes in the pre-image DNF formula.

1.2.1 Input/Output

The program takes Z3 SMT formulas for initial states, transition relation and safety property in no particular normal form, and reports whether the property holds in the system. If enabled, intermediate data structures such as the list of traces, and proof queue at every step can also be displayed.

1.2.2 Limitations

1.2.2.1

The implementation lacks a parser, thus input cannot be provided in SMTLIB format, and need to be typed into the source file directly.

1.2.2.2

The preimgae method distinguishes only between two types of variables, primed and unprimed thus it is not possible to check safety properties of transition systems obtained by loop unrolling, without careful encoding.

2 Proposed method for faster minimal UNSAT core search.

This method takes advantage of the fact that, if a set of constraints is satisfiable then no subset of it is UNSAT.

We start with a cube c with say, n constraints. Check if it satisfies the original query, if it doesn't then we split it into two, say c_1 and c_2, and check it either of them are `UNSAT`. If at least one is, then we recursively minimalize that cube.

If both c_1 and c_2 are `SAT` then we arbitrarily pick a cube, say c_1 and split it into two: a and b. We check if either $a \wedge c_2$ or $b \wedge c_2$ are `UNSAT` and then proceed recursively to obtain the minimal cube c_1'.

At this point, $c_1' \wedge c_2'$ is `UNSAT` but is not a minimal cube. So we, split c_2 into a and b and check if either $a \wedge c_1'$ or $b \wedge c_1'$ are `UNSAT` and then proceed recursively to obtain a minimal cube of c, c'.

This method likely takes $O(nlgn)$ time, but we're unable to work out the details and complete a proof of correctness and complexity at this time.