1 Implementing PDR/IC3 for Linear Integer Arithmetic: Project Proposal.

- Akshay Naik, Devendra Reddy

1.1 Description of the PDR Algorithm

1.1.1 Overall Procedure

```
> PDR(I, T, P): // T, P in CNF
>   F_0 = I
>   n = 1
>   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:
>       s := cube in (F_n && !P)
>       s := generalizeSAT(s)
>       Block(<s,n>)
```

1.1.2 Propagating Phase

```
> Propagate(n):
>   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
>         //Opt?: 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($F_n \wedge \mathrm{T} \wedge \neg P'$) (Optionally, one may optimize the query by adding $\neg s$), 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)
>       r := cube(Pre)
>       r := generalizeSAT(t) //r=>Pre
>       //Why do we need to gen? Why not simply block !Pre ?
>       //(And if needed, remove implied cubes within Pre-img. coz simplify() doesn't do tha
>       Q.add(<r,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
>       //Push forward - Optional
>       for j in [k+1 .. n-1]:
>         if UNSAT(F_(j-1) && !c && T && c'):
>           F_j := F_j && !c
>         else:
>           Q.add(<!c,j>)
>           break
```

```
>   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. If it does, i.e. if `SAT(I && c)` then we need to somehow ungeneralize it minimally. This can be done by `c := c && !I`

1.1.3.2 Pushing c to the latest frame possible

After strengthening all frames up to F_k with $\neg c$, we can further check if it is inductive relative to later frames and block c in those too.

1.1.3.3 Generalizing cube in pre-image from SAT query

Given a pre-image as a set of cubes, we need a generalized cube c such that $c \Rightarrow$`Pre-image`. Is it possible that. `c = True ?`

1.2 Implementation

We intend to implement the entire algorithm in **Python**, using **Z3** for SMT queries. We picked Z3 because we have prior experience with it and it supports unsat core generation and quantifier elimination for linear integer arithmetic. We are yet to investigate how to use it for generalization of cubes from satisfiable queries(for generalizing pre-image).

Each frame is stored as a Z3 Goal object, which can be thought of as a set of formulas. If we restrict each formula to be a clause then each frame can be maintained as a set of clauses. This allows for easy checking of syntactic clause containment and clause propagation. 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.

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.

1.3 Project Deliverables

A Python script implementing the propagation and blocking phases of the PDR algorithm.