Online appendix for the paper

Failure Tabled Constraint Logic Programming by Interpolation

published in Theory and Practice of Logic Programming

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
(*e-mail:* {gkgange,jorge.navas,schachte,harald,pstuckey}@unimelb.edu.au)

Examples with Infinitely Many Failed Derivations

The problem of verifying *safety* properties consists of proving that an unsafe configuration or error is not reachable from an initial configuration considering all possible program executions. CLP has been shown to be a successful model for performing this task (see for example Jaffar et al. (2009) and Angelis et al. (2012)). The program P can be translated into an equivalent CLP program P' such that the error is unreachable if and only if the derivation tree of P' does not contain any successful derivation.

If the derivation tree is finite and the safety property can be expressed, for example, on real numbers then any CLP system can prove the absence of errors. However, programs often contain unbounded loops, and therefore, the main challenge lies in discovering loop invariants that can still prove the unreachability of the error configurations. Another problem for CLP systems is that sometimes the safety property may require reasoning about other theories different from real or rational linear arithmetic.

In this appendix, we show several examples (Figures 1 and 2) taken from the software verification literature which are commonly considered to be challenging for automatic verifiers. We have translated the programs into CLP manually in such way that the original program is safe iff the CLP model of the translated program is empty. The details of the translation are beyond of the scope of this paper and we refer to, for example, Delzanno and Podelski (2001) and Jaffar et al. (2005) for a formal description. We also show the inductive invariant required in each case to prove program is safe.

The only program that cannot be verified by our method is t1.c. Note that a safe inductive invariant is $Y \geq 0 \wedge X \geq Y$. From error/4 we can trivially infer the interpolant $X \geq Y$ but unfortunately we cannot infer the other required invariant $Y \geq 0$ even though that clearly holds, since $Y = 0$ initially and then Y can only be incremented by one. This shortcoming is typical of methods that rely only on counterexample-driven verification with interpolation. However, abstract interpreters using intervals or octagons can easily infer the inductive invariant $Y \geq 0$. Thus, verifiers that combine abstract interpretation with interpolation (Gulavani et al. 2008; Albarghouthi et al. 2012; Jaffar et al. 2012) can easily infer the required invariant $Y \geq 0$.

Note that all C variables are defined as integers. For simplicity, if we would model them as reals then we can still prove all programs are safe except for t4.c since that has

t1.c (from (Angelis et al. 2012))

```
int x=1;
int y=0;
while(*){
  x=x+y, y++;
}
assert(x >= y);
```

t1.pl

Safe inductive invariant: $l(X,Y) \Longrightarrow Y \geq 0 \wedge X \geq Y$

```
t1 :- {X .=. 1, Y .=. 0}, l(X,Y).
l(X,Y):- {X1 .=. X+Y, Y1 .=. Y+1}, l(X1,Y1).
l(X,Y):- error(X,Y).
error(X,Y):- {Y .>. X}.
```

t1-a.c (variant of t1.c)

```
int x=1;
int y=0;
while(*){
  x=x+y, y++;
}
if (y >= 0) assert(x >= y);
```

t1-a.pl

Safe inductive invariant: $l(X,Y) \Longrightarrow Y \geq 0 \wedge X \geq Y$

```
t1 :- {X .=. 1, Y .=. 0}, l(X,Y).
l(X,Y):- {X1 .=. X+Y, Y1 .=. Y+1}, l(X1,Y1).
l(X,Y):- error(X,Y).
error(X,Y):- {Y .<. 0}.
error(X,Y):- {Y .>. X}.
```

t2.c (from (Jhala and McMillan 2006))

```
int x,y,i,j;
x=i; y=j;
while (x != 0){
   x--;
   y--;
}
if (i ==j ) assert(y <= 0);
```

t2.pl

Safe inductive invariant: $l(X,Y,I,J) \Longrightarrow Y + I \leq X + J$

```
t1 :- {X .=. I, Y .=.J}, l(X,Y,I,J).
l(X,Y,I,J):- {X .<>. 0, X1 .=. X-1, Y1 .=. Y-1},
             l(X1,Y1,I,J).
l(X,Y,I,J):- {X .=. 0}, error(X,Y,I,J).
error(_,Y,I,J):- {I .=. J, Y .>. 0}.
```

t3.c (from (Ball et al. 2004))

```
int lock,old,new;
old=0; lock=0; new=old+1;
while (new != old) {
    lock=1;
    old=new;
    if (*) {
       lock=0;
       new++;
    }
}
assert(lock != 0);
```

t3.pl

Safe ind. inv.: $l(Lock, Old, New) \Longrightarrow Old + 1 \leq New$

```
t3:- {Lock .=. 0, Old .=. 0, New .=. Old + 1},
     l(Lock,Old,New).
l(Lock,Old,New):-
     {New .<>. Old},
     l_body(Lock,Old,New,Lock1,Old1,New1),
     l(Lock1,Old1,New1).
l(Lock,Old,New):- { New .=. Old}, error(Lock).
l_body(_Lock,Old,New,Lock1,Old1,New1):-
     {Lock1 .=. 0, New1 .=. New + 1, Old1 .=. Old}.
l_body(_Lock,Old,New,Lock1,Old1,New1):-
     {Lock1 .=. Lock, New1 .=. New, Old1 .=. Old}.
error(Lock):- {Lock .=. 0}.
```

Fig. 1. Verification challenges: C and CLP versions.

(non-integral) answers $\{A = 0, B = 0, I = 0, N > 1, N < 2\}$ and $\{A = 0, B = 0, I = 0, N > 0, N < 1\}$. In this case we can only show the CLP program does not have answers (and hence, proving the program is safe) if we reason over integer linear arithmetic. Since MathSAT provides interpolation for integers we can do this without any effort.

Finally, it is worth pointing out that for t5.c our method needs to unroll the recursive clause of l/2 at least 3 times before it can infer the required invariant. t5.c is a snippet of a real application from the ssh-simplified benchmark suites available at Beyer (2012). This is the only example for which Chico de Guzmán et al. (2012) produced a finite derivation tree without error.

t4.c (from (Beyer et al. 2007))

```
int i,a,b,n;
i=0; a=0; b=0;
assume(n > 0);
while (i < n){
   if (*){
     a=a+1;
     b=b+2;
   }
   else{
     a=a+2;
     b=b+1;
   }
    i++;
}
assert(a+b == 3*n);
```

t4.pl

Safe inductive invariant: $l(I, A, B, N) \Longrightarrow A + B \leq 3 * I$

```
t4 :-  {N .>. 0, I .=. 0, A .=. 0, B .=. 0}
       l(I,A,B,N).

l(I,A,B,N):-
    {I .<. N},
    l_body(A,B,A1,B1),
    {I1 .=. I+1},
    l(I1,A1,B1,N).
l(I,A,B,N):-
    {I .>=.N}, error(A,B,N).

l_body(A0,B0,A1,B1):- {A1 .=. A0+1, B1 .=. B0+2}.
l_body(A0,B0,A1,B1):- {A1 .=. A0+2, B1 .=. B0+1}.

error(A,B,N):- {A + B .<>. 3 * N}.
```

t5.c (snipped from client SSH protocol)

```
int e=0;
int s=2;
while (*) {
  if (s == 2){
    if (e == 0) e=1;
    s=3;
  }
  else if (s == 3){
    if (e == 1) e=2;
    s=4;
  }
  else if (s == 4){
    if (e == 3) error();
    s=5;
  }
}
```

t5.pl

Safe inductive invariant: $l(E, S) \Longrightarrow E \leq 2 \wedge S \leq 4$

```
t5 :- {E .=. 0, S .=. 2}, l(E,S).

l(E0,S0):- l_body(E0,S0,E1,S1), l(E1,S1).
l(E,S)  :- error(E,S).

error(E,S):- {E .=. 3, S .=. 4}.

l_body(E0,S0,E1,S1):-
     {S0 .=. 2}, l_body_1(E0,S0,E1,S1).
l_body(E0,S0,E1,S1):-
     {S0 .=. 3}, l_body_2(E0,S0,E1,S1).
l_body(E0,S0,E1,S1):-
     {S0 .=. 4}, l_body_3(E0,S0,E1,S1).
l_body(E0,S0,E1,S1):-
     {S0 .>. 4,  E1 .=. E0, S1 .=. S0}.

l_body_1(E0,_S0,E2,S2):-
     l_body_1_1(E0,E1), {S2 .=. 3, E2 .=. E1}.
l_body_2(E0,_S0,E2,S2):-
     l_body_2_1(E0,E1), {S2 .=. 4, E2 .=. E1}.
l_body_3(E0,S0,E1,S1):-
     l_body_3_1(E0,E1,S0,S1).

l_body_1_1(E0,E1):- {E0 .=. 0, E1 .=. 1}.
l_body_1_1(E0,E1):- {E0 .<>. 0, E1 .=. E0}.
l_body_2_1(E0,E1):- {E0 .=. 1, E1 .=. 2}.
l_body_2_1(E0,E1):- {E0 .<>. 1, E1 .=. E0}.
l_body_3_1(E0,E1,S0,S1):-
     {S0 .=. 4, E0 .=. 2, E1 .=. E0, S1 .=. S0}.
l_body_3_1(E0,E1,S0,S1):-
     {S0 .<>. 4, S1 .=. 5, E1 .=. E0}.
l_body_3_1(E0,E1,S0,S1):-
     {E0 .<>. 2, S1 .=. 5, E1 .=. E0}.
```

Fig. 2. Verification challenges: C and CLP versions.

References

Albarghouthi, A., Gurfinkel, A., and Chechik, M. 2012. Craig interpretation. In *SAS*. 300–316.

Angelis, E. D., Fioravanti, F., Proietti, M., and Pettorossi, A. 2012. Software model checking by program specialization. In *CILC*. 89–103.

Ball, T., Cook, B., Levin, V., and Rajamani, S. K. 2004. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *IFM*. 1–20.

Beyer, D. 2012. Competition on software verification - (sv-comp). In *TACAS*. 504–524.

Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. 2007. Path invariants. In *PLDI*. 300–309.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., and Stuckey, P. J. 2012. A general implementation framework for tabled CLP. In *FLOPS*. 104–119.

Delzanno, G. and Podelski, A. 2001. Constraint-based deductive model checking. *STTT 3,* 3, 250–270.

Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. 2008. Automatically refining abstract interpretations. In *TACAS*. 443–458.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV*. 758–766.

Jaffar, J., Santosa, A. E., and Voicu, R. 2005. Modeling systems in clp. In *ICLP*. 412–413.

Jaffar, J., Santosa, A. E., and Voicu, R. 2009. An interpolation method for CLP traversal. In *CP*. 454–469.

Jhala, R. and McMillan, K. L. 2006. A practical and complete approach to predicate refinement. In *TACAS*. 459–473.