ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement

Andreas Podelski[1,3] and Andrey Rybalchenko[2,3]

[1] University of Freiburg
[2] Ecole Polytechnique Fédérale de Lausanne
[3] Max-Planck-Institut für Informatik Saarbrücken

Abstract. Software model checking with abstraction refinement is emerging as a practical approach to verify industrial software systems. Its distinguishing characteristics lie in the way it applies logical reasoning to deal with abstraction. It is therefore natural to investigate whether and how the use of a constraint-based programming language may lead to an elegant and concise implementation of a practical tool. In this paper we describe the outcome of our investigation. Using a Prolog system together with Constraint Logic Programming extensions as the implementation platform of our choice we have built such a tool, called ARMC (for Abstraction Refinement Model Checking), which has already been used for practical verification.

1 Introduction

Software model checking with (counterexample-guided) abstraction refinement is emerging as a practical approach to verify industrial software systems [2,4,5,13,16]. Its distinguishing characteristics lie in the way it applies logical reasoning to deal with abstraction. In particular, it implements the automatic construction of abstract domains based on logical formulas. This construction requires intricate operations on logical formulas, operations which involve both syntax-based manipulations and semantics-based logical operations such as entailment tests between constraints. It is therefore natural to investigate whether and how the use of a constraint-based logic programming language may lead to an elegant and concise implementation of a practical tool. In this paper we describe the outcome of our investigation.

Using a Prolog system together with extensions [15,17] as the implementation platform of our choice we have built such a tool, called ARMC (for Abstraction Refinement Model Checking). The tool has already been used for practical verification [20].

Our work builds upon, and also crucially differs from previous efforts to exploit constraint based programming languages for the implementation of model checkers (see e.g. [1,8,9,10,11,18,19,21]). Those efforts relate the fixpoint definitions of runtime properties of programs with the fixpoint semantics of

M. Hanus (Ed.): PADL 2007, LNCS 4354, pp. 245–259, 2007.

constraint logic programs. We also take advantage of this connection, but our implementation may best be understood by its operational reading. We exploit the logical reading of programming language constructs for the implementation of operations that are specific to abstraction and abstraction refinement. As far as we know, none of the existing CLP/logic-based implementations of model checkers performs abstraction refinement.

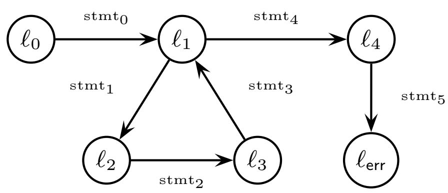
We structure the paper as follows. First, we describe the representation of the program to be verified by Prolog facts `trans(...)` that are stored in the Prolog database. We then define the procedure `post` that implements the one-step-reachability operator over sets of states, each set being represented by a constraint. The abstraction procedure `abstract` takes a set of predicates (which are atomic constraints stored in the Prolog database in a single fact `preds(...)`) and maps a set of states to the corresponding over-approximation. We define `abstract`, `concretize` and `abstract_post`. We are then ready to define the abstract reachability procedure `abstract_fixpoint`.

If the abstraction is too coarse then the call to `abstract_fixpoint` may lead to the call of a refinement procedure `refine`, which updates the Prolog fact `preds(...)` stored in the Prolog database. The subsequent iteration calls the abstract reachability procedure again, but now the procedure `abstract` refers to the new set of predicates. The refinement procedure is based on the procedure `feasible` that performs an intricate analysis of counterexamples that are possible in the abstract, but may be absent in the concrete. The insights that are gained during this analysis guide the discovery of new predicates which are added in order to refine abstraction (for a detailed account on the underlying algorithm we refer to [3]). We first define the procedure `feasible` and then `refine`, and are then finally ready to define the 'main' procedure ARMC, which is `abstract_check_refine`.

2 From Program Statements to Prolog Facts `trans(...)`

We illustrate the translation of the program to be verified into the representation by Prolog facts in Figure 1. We translate each statement of the corresponding goto program by a `trans(...)`-fact (all `trans(...)`-facts together represent the transition relation of the program to be verified). In the next section, we will use calls of the form `trans(FromState, ToState, Rho, StmtId)` where the first two arguments represent the states (control location and data variables) before and after the execution of the statement. The third argument will be bound to a term that stands for a *transition constraint*, e.g. `Rho = (Xp=X+1, Yp=Y)`. Here the logical variables `X` and `Xp` (read "x-prime") refer to the before- and after-values of the C program variable `x`. Transition constraints relate the values of program variables before and after the transition. We use the expression language of the applied CLP system to form transition constraints. The fourth argument will be bound to the label that identifies the statement. We encode the initial and error conditions of the program with the help of the distinguished locations `start(...)` and `error(...)`.

```
0:  int x=0, y=0;
1:  while ( x =< 10 ) {
2:    x=x+1;
3:    y=y+1;
    }
4:  assert( x==y );
```




```
start(ctrl(loc_0)).

error(ctrl(loc_err)).

trans(s(ctrl(loc_0), data(X, Y)),  s(ctrl(loc_1), data(Xp, Yp)),         (Xp=0, Yp=0), stmt_0).

trans(s(ctrl(loc_1), data(X, Y)),  s(ctrl(loc_2), data(Xp, Yp)),   (X=<10, Xp=X, Yp=Y), stmt_1).

trans(s(ctrl(loc_2), data(X, Y)),  s(ctrl(loc_3), data(Xp, Yp)),       (Xp=X+1, Yp=Y), stmt_2).

trans(s(ctrl(loc_3), data(X, Y)),  s(ctrl(loc_1), data(Xp, Yp)),       (Xp=X, Yp=Y+1), stmt_3).

trans(s(ctrl(loc_1), data(X, Y)),  s(ctrl(loc_4), data(Xp, Yp)),    (X>10, Xp=X, Yp=Y), stmt_4).

trans(s(ctrl(loc_4), data(X, Y)), s(ctrl(loc_err), data(Xp, Yp)),  (X=\=Y, Xp=X, Yp=Y), stmt_5).
```

Fig. 1. Example program in C syntax and its representation by Prolog facts. The correctness of the program is defined by the validity of the assertion in line 4. In terms of the corresponding goto program depicted by the control-flow graph this means the non-reachability of the error location ℓ_{err} from the start location ℓ_0. It is always possible to encode the initial and the error condition of the program with the help of special locations ℓ_0 and ℓ_{err}.

3 One-Step-Reachability Operator `post`

Figure 2 shows the procedure `post` that implements the one-step-reachability operator over sets of states.

We "symbolically" represent a set of states by a constraint. For example, the constraint `Y>=5, X=Y` represents the set of all valuations of the program variables (see Figure 1) where the program variable `y` is not less than 5 and is equal to the value of the program variable `x`. A program state is determined by the valuation of the program variables and the control location. Assume the bindings `Phi = (Y>=5, X=Y)`, and `FromState = s(ctrl(loc_2), data(X, Y))`. Then `Phi` and `FromState` together represent the set of program states at the location ℓ_2 with the valuations of the program variables constrained as described above. We explain the use of the `data(...)` term later.

We consider the set of successor states under the execution of a particular program statement in the goto program. The forth parameter of `post` is used to identify this statement. In our example, the identifiers of statements, i.e. the possible values of `StmtId`, range from `stmt_0` to `stmt_5`.

We use our example to illustrate how `post` is executed. Assume the above bindings for `Phi` and `FromState`. The call `{Phi}` injects the constraint `Y>=5, X=Y` into the constraint store. The next call non-deterministically selects a `trans(...)` fact from the database, say the fact identified by `stmt_2`. This creates the bindings

```
ToState = s(ctrl(loc_3), data(Xp, Yp)),
Rho = (Xp=X+1, Yp=Y),
StmtId = stmt_2.
```

We observe that the variables in the term bound to `FromState` are unified with the from-variables of the transition. In the example, for legibility, we have already chosen the same variables, i.e., `X` and `Y` both for the variables in `FromState` and for the from-variables.

The call `{Rho}` injects the transition constraint `Xp=X+1, Yp=Y` into the constraint store. This means that the constraint store now contains the constraint `Y>=5, X=Y, Xp=X+1, Yp=Y`. The projection of this constraint on the variables `Xp` and `Yp` represents the set of valuations of the program variables after the application of the statement identified by `StmtId`. This projection yields `Xp=1+Yp, Yp>=5`. It is instructive to reflect that this constraints indeed represents the successor values of `x` and `y` after the increment operation for `x`.

The choice of the variables for the projection is determined by the term bound to `ToState`, which is `s(ctrl(loc_3), data(Xp, Yp))` in our example. The projection is performed by the elimination of existentially quantified variables, in the example `X` and `Y`. We do not explicitly perform this elimination (neither the renaming of primed by unprimed variables, which is usually required by implementations of successor operators).

```
post(Phi, FromState, ToState, StmtId) :-
        {Phi},
        trans(FromState, ToState, Rho, StmtId),
        {Rho}.
```

Fig. 2. The procedure `post`

4 Abstract One-Step Reachability Operator `abstract_post`

The procedure `abstract_post` implements a function that is defined by the functional composition of three functions for which the notation α, *post* and γ is customary in the abstract interpretation framework [7]: the abstraction, the one-step-reachability operator, and the concretization. As we will show below, the procedure `abstract_post` is implemented in terms of the three procedures `abstract`, `post` and `concretize`.

Procedure `abstract`. We define the procedure `abstract` in Figure 3. This procedure computes a constraint that is an over-approximation of the current content of the constraint store. The first argument of `abstract` determines the approximation function. For example, `Xp=1+Yp, Yp>=5` is approximated by the constraint `Yp>=0, Xp>=Yp` if the list of the four constraints `Xp=<0, Yp>=0, Xp=<Yp, Xp>=Yp` appears in the first parameter of `abstract`. It is customary to refer to the given set of constraints (which together determine

```
abstract([Pred-Id|PredIdPairs], Ids) :-
        ( entailed(Pred) ->
            abstract(PredIdPairs, TmpIds),
            Ids = [Id|TmpIds]
        ;
            abstract(PredIdPairs, Ids)
        ).
abstract([], []).
concretize([Id|Ids], [Pred-PId|PredIdPairs], Phi) :-
        ( Id = PId ->
            concretize(Ids, PredIdPairs, TmpPhi),
            Phi = (Pred, TmpPhi)
        ;
            concretize([Id|Ids], PredIdPairs, Phi)
        ).
concretize([], _, 1=1).
abstract_post(FromCtrl, FromIds, ToCtrl, ToIds, StmtId) :-
        FromState = s(FromCtrl, _),
        preds(FromState, FromPredIdPairs),
        concretize(FromIds, FromPredIdPairs, Phi),
        post(Phi, FromState, ToState, StmtId),
        ToState = s(ToCtrl, _),
        preds(ToState, ToPredIdPairs),
        abstract(ToPredIdPairs, ToIds).
```

Fig. 3. The procedures `abstract`, `concretize`, and `abstract_post`

the approximation function) as *predicates*. In our running example, we refer to the four predicates given above.

We give each predicate a unique identifier. This is its position in a given list of predicates. The call `abstract(PredIdPairs, Ids)` computes a list of identifiers that is bound to `Ids`. This list consists of the identifiers of the predicates that appear in the approximation of the constraint in the constraint store. For technical reasons, the first parameter of `abstract` is not a list of predicates, but a list of pairs containing a predicate and its identifier (which we write using `-` in Prolog).

We continue our example. If `PredIdPairs` is bound to `[(Xp=<0)-1, (Yp>=0)-2, (Xp=<Yp)-3, (Xp>=Yp)-4]` and the constraint store contains `Xp=1+Yp, Yp>=5` then `abstract` creates the binding `Ids = [2,4]`.

Note that we have used an implicit assumption. Namely, the variables that appear in the constraint to be approximated are literally the variables that appear in the list of predicates (from predicate-identifier pairs). This assumption is justified by the context in which `abstract` is called. Namely, the call `abstract(PredIdPairs, Ids)` is preceded by the call `preds(State, PredIdPairs)` and `State` is bound to a term of the form `s(..., data(Xp,Yp))`.

We assume that the Prolog database contains a fact of the form `preds(...)`. In our example, this fact is

```
preds(s(ctrl(_), data(X, Y)), [(X=<0)-1, (Y>=0)-2, (X=<Y)-3, (X>=Y)-4]).
```

The call `preds(State, PredIdPairs)` now succeeds and realizes the appropriate α-renaming in the predicates, namely by unifying the variable `X` and `Y` with `Xp` and `Yp` respectively. Therefore it computes the binding of `PredIdPairs` shown above.

Procedure `concretize`**.** The procedure `concretize` is defined in Figure 3. It takes a list of identifiers and computes a constraint that is the conjunction of predicates whose identifiers are in the input list. As `abstract`, the procedure `concretize` takes a list of predicate-identifier pairs as a parameter. Continuing our example, we call `concretize(Ids, PredIdPairs, Phi)` given the binding of `Ids` to the list of predicate identifiers `[2, 4]` and the above binding of `PredIdPairs`. The resulting binding to `Phi` is `Yp>=0, Xp>=Yp, 1=1`.

Procedure `abstract_post`**.** The procedure `abstract_post` is given in Figure 3. It is the composition of the procedures `concretize`, `post`, and `abstract`.

We may view the procedure `abstract_post` as a function that maps an abstract state to a successor abstract state (for a fixed statement). We define an abstract state as the pair given by a control location and a list of identifiers of predicates. For example, under the binding of `FromCtrl` to `ctrl(loc_2)` and the binding of `FromIds` to the list of identifiers `[2, 4]`, an abstract state is given by `FromCtrl` and `FromIds`.

The application of `abstract_post` on `FromCtrl` and `FromIds` under the above binding computes a successor abstract state as follows. The execution of the first line binds `FromState` to the term `s(ctrl(loc_2), FromData)` where `FromData` is a fresh variable. The call `preds(FromState, FromPredIdPairs)` binds the list of predicate-identifier pairs that is stored in the Prolog database to `FromPredIdPairs`. These predicates are over fresh variables, say `X` and `Y`. The variable `FromData` gets bound to the term `data(X, Y)`.

Now, the call to `concretize` translates the list of predicate identifiers `[2, 4]` to the constraint `Y>=0, X>=Y, 1=1`, which is bound to `Phi` (and represents the set of states whose successors will be computed and abstracted).

The call of the procedure `post` proceeds as described in Section 3. We assume that the statement `stmt_2` is selected for application. This statement goes from location ℓ_2 to location ℓ_3. The call to `post` binds `ToState` to the term `s(ctrl(loc_3), data(Xp, Yp))`, where `Xp` and `Yp` are fresh variables. Now, the constraint store contains the constraint `Y>=0, X>=Y, 1=1, Xp=X+1, Yp=Y`. Its projection to the variables `Xp` and `Yp` that are referenced by `ToState` is a new constraint, namely, `Xp>=1+Yp, Yp>=0`. It represents the set of states that are reachable by applying the statement `stmt_2` to the set of states denoted by the constraint `Y>=0, X>=Y, 1=1` (which is the previously computed concretization of the abstract state given by `FromState` and `FromIds`).

```
assert_abst_reach_state(_, Ctrl, Ids, _, _, _) :-
        abst_reach_state(_, Ctrl, ReachedIds, _),
        ord_subset(ReachedIds, Ids),
        !.
assert_abst_reach_state(Iter, Ctrl, Ids,
                        AbstStateId, StmtId, NextAbstStateId) :-
        bb_get(abst_reach_state_count, LastAbstStateId),
        NextAbstStateId is LastAbstStateId+1,
        bb_put(abst_reach_state_count, NextAbstStateId),
        assert(abst_reach_state(iter(Iter),Ctrl,Ids,NextAbstStateId)),
        assert(abst_parent(NextAbstStateId, from(state(AbstStateId),
                                             trans(StmtId))))).
abstract_fixpoint_step(Iter, NextIter) :-
        abst_reach_state(iter(Iter), FromCtrl, FromIds, AbstStateId),
        abstract_post(FromCtrl, FromIds, ToCtrl, ToIds, StmtId),
        assert_abst_reach_state(NextIter, ToCtrl, ToIds,
                                AbstStateId, StmtId, NextAbstStateId),
        ( error(ToCtrl) ->
            throw(abst_error_state(NextAbstStateId))
        ;
            true
        ).
abstract_fixpoint(Iter) :-
        NextIter is Iter+1,
        ( bagof(_, abstract_fixpoint_step(Iter, NextIter), _) ->
            abstract_fixpoint(NextIter)
        ;
            true
        ).
```

Fig. 4. The procedures `assert_abst_reach_state`, `abstract_fixpoint_step`, and `abstract_fixpoint`. `bb_get`/`bb_put` store/read facts from the mutable repository.

The execution of `ToState = s(ToCtrl, _)` binds `ToCtrl` to the term `ctrl(loc_3)`, which represents the to-location. The call to `abstract` assumes that it is applied to the predicates over the variables `Xp` and `Yp`. We create such predicates by calling `preds` with the first parameter bound to `s(ctrl(loc_3), data(Xp, Yp))`. Finally, the outcome of the call to `abstract` is a list of predicate identifiers `[2, 4]` that is bound to `ToIds`.

5 Abstract Reachability Procedure `abstract_fixpoint`

We define the procedure `abstract_fixpoint` together with the auxiliary procedures `assert_abst_reach_state`, `abstract_fixpoint_step` in Figure 4.

Figure 8 (shown in the appendix) presents the execution of `abstract_fixpoint` on our example program, which is shown in Figure 1. The procedure `abstract_fixpoint` computes an approximation of the set of reachable states of the program to be verified. It also checks whether the error location is contained in the approximation, i.e., if an abstract state at location `loc_err` is created. If this check succeeds then the iteration halts and throws an exception. We discuss the exception handling in Section 6.

The procedure `abstract_fixpoint` implements a fixpoint computation that iteratively builds up a set of facts `abst_reach_state(...)` stored in the Prolog database. Each such fact represents an abstract state that is determined to be reachable by the abstract fixpoint computation. For example, the fact `abst_reach_state(iter(2), ctrl(loc_2), [2,3], 3)` represents an abstract state at the control location `ctrl(loc_2)` and the list of predicate identifiers `[2, 3]`. The first argument of `abst_reach_state(...)`, here `iter(2)`, shows at which iteration the abstract state is created and inserted into the database. The last argument shows the identifier of the abstract state, which is `3` in our example. Since the list `[2, 3]` refers to the predicates `X-Y=<0, X-Y>=0` (from the list of predicates as fixed by the fact `preds(...)` currently in the Prolog database, see Figure 8), the abstract state represents the set of program state at the location ℓ_3 with equal values of the variables `x` and `y`. Figure 8 also shows facts `abst_parent(...)`. We do not discuss them in this section. They will play a role in Section 6.

The procedure `assert_abst_reach_state` first checks whether a given abstract state, which is represented by `Ctrl` and `Ids`, is already present in the database. This is the case if there exists a reachable abstract state whose list of identifiers `ReachedIds` is contained in the list `Ids`. In this case the given abstract state represents a smaller set of program states at the same control location. For example, an abstract state with predicate identifiers `[2, 3, 4]` represents a smaller set of program states than an abstract state with predicate identifiers `[3, 4]`. A longer list of identifiers corresponds to a larger conjunction of predicates, i.e. to a stronger constraint. We implement the comparison between lists of identifiers by a call to the library procedure `ord_subset` because our implementation guarantees that these lists are ordered.

The procedure `assert_abst_reach_state` inserts the given abstract state into the database if it is not already present. It computes the value for `NextAbstStateId`, which is used to label the given abstract state.

The procedure `abstract_fixpoint` calls `abstract_fixpoint_step` by using the `bagof` procedure of Prolog. It iterates over all abstract states that are created at the iteration with number `Iter` (and stored as `abst_reach_state(...)` facts in the Prolog database) and over all program statements (which are stored as `trans(...)` facts). The call to `abstract_fixpoint_step` fails if no new abstract state is created (and hence a fixpoint is reached).

6 Abstraction Refinement Procedure `abstract_check_refine`

Given a set of predicates, the procedure `abstract_fixpoint` computes an over-approximation of the reachable state space of the program, as we described in the previous section. If this over-approximation does not contain the error location then the program is proven correct. Otherwise, there exists a sequence of abstract states that begins at the start location and ends at the error location. Each step in this sequence corresponds to the application of a program statement to an abstract state. We call this sequence of statements a *counterexample path*, or *counterexample* for short. Now, the procedure `feasible` determines which of the following two cases applies.

In the first case, the error location is indeed reachable (from the initial location) by executing the sequence of statements. We say that the counterexample is *feasible*. We report that the program is not correct and return the counterexample. In the second case, the sequence is not feasible. We say that the counterexample is *spurious*. The abstraction was too coarse. This means that the set of predicates does not yet contain the "right" predicates. The procedure `refine` discovers new predicates and adds them to the set of existing ones.

The procedure `abstract_check_refine` repeatedly executes `abstract_fixpoint`, `feasible`, and `refine`. It terminates in one of two cases. Either a feasible counterexample is computed, or it discovers the right set of predicates. The latter case means that the procedure `abstract_fixpoint` computes a sufficiently precise over-approximation of the set of reachable states of the program, one which does not contain the error location. In this section, we define the procedures `feasible`, `refine`, and `abstract_check_refine`.

Counterexample checking procedure `feasible`. We check the feasibility of the path between the initial and error location in the abstract reachability tree by applying the procedure `feasible`. It is defined in Figure 5. If the procedure succeeds for the abstract state identifier `SId` that is given in the exception `abst_error_state(ErrorStateId)`, see Figure 4, then we report that the program is incorrect and print the error path.

```
feasible(AbstStateId, ToState, AccPath, ErrorPath) :-
        ( abst_parent(AbstStateId, from(state(PrevAbstStateId),
                                        trans(StmtId))) ->
            trans(FromState, ToState, Rho, StmtId),
            {Rho},
            feasible(PrevAbstStateId,FromState,[StmtId|AccPath],ErrorPath)
        ;
            ErrorPath = AccPath
        ).
```

Fig. 5. The procedure `feasible`

Continuing our example, we will follow the execution of the call `feasible(9, _, [], ErrorPath)`. We assume the context of Figure 8. That is, the call of `abstract_fixpoint` has inserted the shown `abst_parent(...)` facts. These facts form a tree whose root is the start abstract state `0`. Each path in the tree corresponds to a sequence of statements, according to the `abst_parent(...)` facts. The call `feasible(9, _, [], ErrorPath)` determines whether the path is feasible or whether it is a spurious counterexample.

The first execution step of the call `feasible(9, _, [], ErrorPath)` retrieves the fact `abst_parent(9, from(state(8), trans(stmt_5)))` and binds `StmtId` to `stmt_5`. Then, it retrieves the fact

```
trans(s(ctrl(loc_4), data(X1, Y1)), s(ctrl(loc_err), data(X0, Y0)),
      (X1=\=Y1, X0=X1, Y0=Y1), stmt_5)
```

and binds `Rho` to the transition constraint `X1=\=Y1, X0=X1, Y0=Y1`. The next line injects this constraint into the constraint store.

The effect of the recursive call to `feasible` is that the line `{Rho}` in that recursive call injects the transition constraint `X2>10, X1=X2, Y1=Y2`, which belongs to the statement `stmt_4`. This statement precedes the statement `stmt_5` on the path that ends in the abstract state `9`.

The recursion in the procedure `feasible` terminates, and upon termination we distinguish two cases. In the first case, the conjunction of transition constraints that are injected into the constraint store is not satisfiable. This means that the corresponding sequence of statements is not feasible. In the second case, we have explored the path from the given abstract state to the start abstract state. Since the start abstract state does not have a corresponding `abst_parent` fact, the call `abst_parent(1, ...)` fails. Hence, `feasible` terminates and binds `ErrorTrace` to the list of identifiers of the statements along the path.

In our example, the call `feasible(9, ...)` fails. The transition constraint for the statement `stmt_0` is inconsistent with the conjunction of the transition constraints for other statements on the path leading to the error abstract state `9`. This means that the call `{Rho}` fails in the recursive call `feasible(2, ...)`.

We have already discussed the handling of fresh variables in terms `FromState` and `ToState` in Section 3. The situation here is analogous. We need to create instances of constraints over the appropriate variables. We observe that the term bound to `FromState` gets passed to the formal parameter `ToState` in the recursive call to `feasible`. Hence, we obtain the sequence of transition constraints such that the from-variables of each constraint are equal to the to-variables of its successor constraint. In our example, the constraint store contains `X1=\=Y1, X0=X1, Y0=Y1,  X2>10, X1=X2, Y1=Y2` after the first recursive call to `feasible`.

Predicate discovery procedure `refine`. The procedure `refine` is defined in Figure 6. We assume that each transition constraint can be partitioned into two lists. The first list consists of constraints over from-variables, and is called list of guards. The second list consists of a list of update expressions of the form `Xp = Exp` where `Xp` is a to-variable and `Exp` is an expression over the from-variables.

```
wp(Updates, Guards, Formula, WP) :-
        ( Updates = [U|Us] ->
            U,
            wp(Us, Guards, Formula, WP)
         ;
            append(Guards, Formula, WP)
         ).
refine(AbstStateId, ToState, Formula) :-
        ( abstract_parent(AbstStateId, from(state(PrevAbstStateId),
                                            trans(StmtId))) ->
            stmt(FromState, ToState, Guards, Updates, StmtId),
            wp(Updates, Guards, Formula, WP),
            insert_preds(FromState, WP),
            refine(PrevAbstStateId, FromState, WP)
        ;
            true
        ).
```

Fig. 6. The procedures `wp` and `refine`

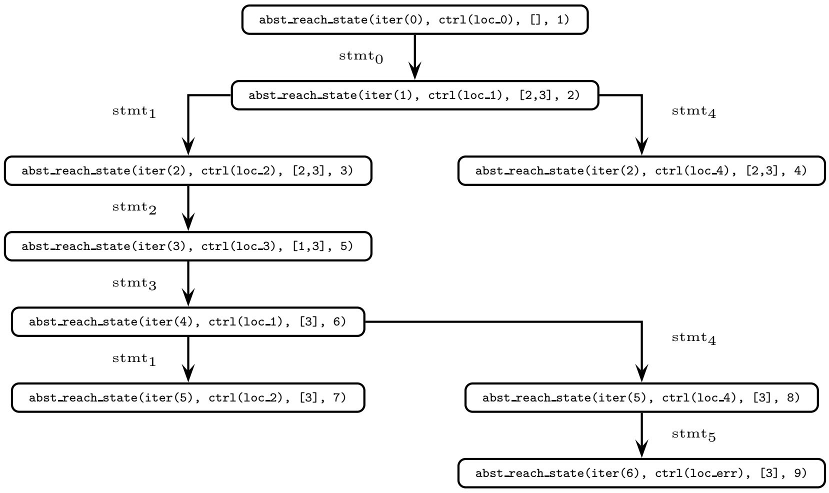
```
abstract_check_refine :-
        start(StartCtrl),
        bb_put(abst_reach_state_count, 1),
        assert(abst_reach_state(iter(0), StartCtrl, [], 1)),
        catch( abstract_fixpoint(0),
               abst_error_state(AbstErrorStateId),
               ( feasible(AbstErrorStateId, _, [], Path) ->
                   format('counterexample ~p\n', [Path]),
                   fail
               ;
                   refine(AbstErrorStateId, _, []),
                   retractall(abst_reach_state(_, _, _, _)),
                   retractall(abst_parent(_, _)),
                   abstract_check_refine
               )
             ).
```

Fig. 7. The procedure `abstract_check_refine`

For each fact `trans(FromState, ToState, Rho, StmtId)` we assume that the Prolog database contains a fact `stmt(...)` of the form

```
stmt(FromState, ToState, Guards, Updates, StmtId)
```

where `Guards` and `Updates` form a partition of `Rho`. For example, given the bindings `FromState = s(ctrl(loc_4), data(X, Y))`, `ToState = s(ctrl(loc_err), data(Xp, Yp)`, and `StmtId = stmt_5` we obtain the list of guards `[X=\=Y]` and the list of updates `[Xp=X, Yp=Y]`.




```
preds(s(ctrl(_), data(X,Y)), [(X-Y>=1)-1, (X-Y=<0)-2, (X-Y>=0)-3]).
```

```
abst_reach_state(iter(0), ctrl(loc_0),   [],    1).
abst_reach_state(iter(1), ctrl(loc_1),   [2,3], 2).
abst_reach_state(iter(2), ctrl(loc_2),   [2,3], 3).
abst_reach_state(iter(2), ctrl(loc_4),   [2,3], 4).
abst_reach_state(iter(3), ctrl(loc_3),   [1,3], 5).
abst_reach_state(iter(4), ctrl(loc_1),   [3],   6).
abst_reach_state(iter(5), ctrl(loc_2),   [3],   7).
abst_reach_state(iter(5), ctrl(loc_4),   [3],   8).
abst_reach_state(iter(6), ctrl(loc_err), [3],   9).
```

```
abst_parent(2, from(state(1), trans(stmt_0))).
abst_parent(3, from(state(2), trans(stmt_1))).
abst_parent(4, from(state(2), trans(stmt_4))).
abst_parent(5, from(state(3), trans(stmt_2))).
abst_parent(6, from(state(5), trans(stmt_3))).
abst_parent(7, from(state(6), trans(stmt_1))).
abst_parent(8, from(state(6), trans(stmt_4))).
abst_parent(9, from(state(8), trans(stmt_5))).
```

Fig. 8. The facts `abst_reach_state(...)` and `abst_parent(...)` computed and asserted by the call of `abstract_fixpoint`. We assume the context of the Prolog database with the given fact `preds(...)` (fixing the set of predicates) and the `trans(...)`-facts given in Figure 1 (representing the program to be verified). The pictorial representation relates the facts `abst_reach_state(...)` by edges according to the facts `abst_parent(...)`.

We continue our example. We follow the execution of the call `refine(9, _, [])`. This call is performed after the call `feasible(9, ...)` fails. The call to `abstract_parent` binds `PrevAbstStateId` to `8` and `StmtId` to `stmt_5`. The next line retrieves the guards and updates for `stmt_5`. These are passed to the procedure `wp`, which computes the weakest precondition of `Formula` with respect to the guards and updates.

The call `wp([Xp=X, Yp=Y], [X=\=Y], [], WP)` binds `WP` to `[X=\=Y]`. The call to `insert_preds(s(ctrl(loc_4), data(X, Y)), [X=\=Y])` adds the predicates to the list of predicates that is stored in the Prolog database as `preds(...)`. The recursive call to `refine` continues the discovery of predicates, which is guided by the remaining statements from the counterexample.

We continue to follow the execution of `refine` and show the execution of the call to `wp` after the second recursive step. For simplicity of presentation

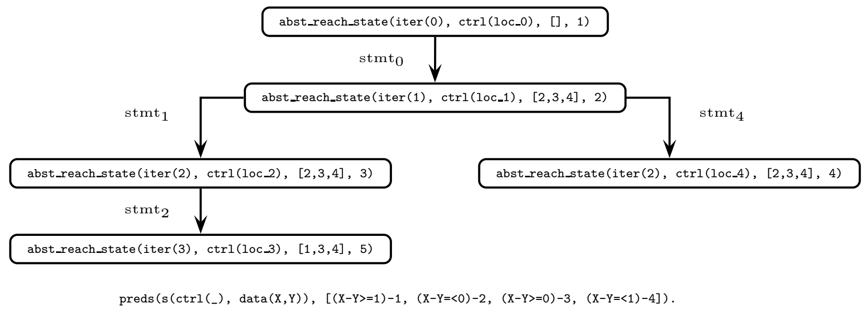


Fig. 9. Sufficiently precise reachable abstract states computed by `abstract_check_refine` for the program in Figure 1. None of abstract states visits the error location `ctrl(loc_err)`.

we assume the from-variables `X` and `Y` together with to-variables `Xp` and `Yp`. Then, the call `wp([Xp=X, Yp=Y+1], [], [Xp=\=Yp], WP)` binds `WP` to the list `[X=\=Y+1]`.

The presented implementation of `WP` exploits the particular syntactic form of update expressions, and can be generalized to arbitrary updates by resorting to the projection of the constraint store, e.g. using techniques from [12].

Abstraction refinement procedure `abstract_check_refine`. The procedure `abstract_check_refine` is defined in Figure 7. It calls the procedures `abstract_fixpoint`, `feasible`, and `refine` as described above.

We continue the illustration based on the example in Figure 1. See Figure 8. First, `abstract_check_refine` creates the root of the tree. It binds `StartCtrl` to the start location. For our program it is `loc_0`. Then, it initializes the counter for reachable abstract states. The creation of the start abstract state completes the setup required to compute the reachable abstract states. Now, the abstract reachability tree is computed by `abstract_fixpoint`. The control location of the abstract state `9` is the error location. Hence, after this abstract is created the procedure `abstract_fixpoint` throws an exception given by the term `abst_error_state(9)`. This exception triggers the analysis of the corresponding counterexample by the procedure `feasible`. The analysis is described above in this section. Its outcome is negative, i.e., `feasible` fails. The call to `refine` refines the abstraction. Now, the previously created facts `abst_reach_state` and `abst_parent` are pruned from the Prolog database. This finishes the current iteration of `abstract_check_refine`.

We continue with the recursive call to `abstract_check_refine`. See Figure 9. It shows the new set of predicates computed by the refinement procedure. Again, the root of the tree is created and the tree is computed by a call to `abstract_fixpoint`. Observe that the error location `loc_err` is not reached. ARMC proves the program correct.

7 Conclusion and Future Work

By presenting the procedures above, we have demonstrated how the use of a constraint-based logic programming language may lead to an elegant and concise implementation of a practical tool for software model checking with abstraction refinement.

We believe that our work may trigger further activities of research in two directions, corresponding to two groups of researchers. The first group consists of expert logic programmers who can optimize the presented implementation by using the programming constructs we have found suitable, but doing so in more sophisticated ways than we have been able to. The second group consists of expert developers of software verification tools who want to evaluate new algorithms (e.g. for abstraction refinement) and use the implementation techniques that we present in this paper.

Acknowledgements. We thank Jan-Georg Smaus for his comments on the paper.

References

1. E. Albert, P. Arenas-Sánchez, G. Puebla, and M. V. Hermenegildo. Reduced certificates for abstraction-carrying code. In *ICLP*. 2006.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*. 2001.
3. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*. 2002.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*. 2003.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*. 2003.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*. 2000.
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. B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *PLILP*. 1998.
9. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*. 1999.
10. C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1-3), 2004.
11. L. Fribourg. Constraint logic programming applied to model checking. Invited tutorial. In *LOPSTR*. 2000.
12. N. Heintze, S. Michaylov, P. Stuckey, and R. Yap. Meta-programming in CLP(R). *J. of Logic Programming*, 33(3), 1997.
13. T. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy abstraction. In *POPL*. 2002.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*. 2004.
15. C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

16. F. Ivancic, H. Jain, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*. 2005.
17. J. Jaffar and J. Lassez. Constraint logic programming. In *POPL*. 1987.
18. J. Jaffar, A. E. Santosa, and R. Voicu. A CLP method for compositional and intermittent predicate abstraction. In *VMCAI*. 2006.
19. M. Leuschel and M. Butler. Combining CSP and B for specification and property verification. In *FM*. 2005.
20. R. Meyer, J. Faber, and A. Rybalchenko. Model checking data-expensive real-time systems. To appear in *ICTAC*, 2006.
21. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *CL*. 2000.
22. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. Submitted, 2006.