NuSMV 2.2.2 modifications to support AR-CTL

Franco Raimondi

Abstract

Technical details of the implementation of Action-restricted logic (AR-CTL) operators into NuSVM. See Charles' notes on AR-CTL.

1 New Syntax

- I defined the following new keywords:
  `EAX, AAX, EA, AA, EAF, AAF, EAG, AAG, REACHABLE, START`
- New grammar rules:

```
  EAX simple_expr ctl_expr
| AAX simple_expr ctl_expr
| EAF simple_expr ctl_expr
| AAF simple_expr ctl_expr
| EAG simple_expr ctl_expr
| AAG simple_expr ctl_expr
| EEA simple_expr TOK_LB ctl_expr TOK_UNTIL ctl_expr TOK_RB
| AAA simple_expr TOK_LB ctl_expr TOK_UNTIL ctl_expr TOK_RB
| REACHABLE simple_expr
| START
```

2 Overview of the changes

Apart from defining the new grammar (see above) files
`parser/input.l, parser/symbol.h, parser/grammar.y`,
I changed the following files to support the new node types:
`compile/compileFlatten.c, compile/compileCone.c, node/nodeWffPrint.c`.

Also, I disabled a check in `prop/propDb.c` to allow IVAR's to appear in `ctl_expr`. **THIS MUST BE RE-INTRODUCED** at some point. I think that a new procedure must be created, to recursively check the structure of the formula, so that `simple_expr` is an IVAR-only expression.

2.1 Model checking procedure for the new operators

I tried to keep the same coding style for the new operators. In `mc/mcEval.c`, I introduced new `case:` statements for the new operators. I introduced the following new functions in `mc/mcMc`: `eax, eau, aau, eaf, eag, reachable`.

I could use the existing `binary_mod_bdd_op` for the "unary" action operators (e.g. EAX, AAF, etc.), but I had to introduce `binary_act_bdd_op` for `eau` and `aau`. For the unary operators, IVAR expressions are evaluated in the `default` case of `eval_spec_recur` via `BddEnc_expr_to_bdd`. Then, the BDD's for IVAR's are used in the verification. For example, for EAX:

```
BddStates eax(BddFsm_ptr fsm, BddStates f, BddStates g)
{
/* Here f is an IVAR expression */
  bdd_ptr tmp_1, result;
  tmp_1 = bdd_and(dd_manager, f, g);
  result = ex(fsm, tmp_1);
  bdd_free(dd_manager, tmp_1);
  return(result);
}
```

For EAU and AAU: `binary_act_bdd_op` computes the BDD for the two CTL expression and for the IVAR expression, and then `eau` or `aau` are called with these three parameters. Notice: I had to add a new type BDDPFFBBB for operators with 3 bdd vars (see mcEval.c).

3 List of modified files

```
compile/compileFlatten.c
compile/compileCone.c
mc/mc.h
mc/mcEval.c
mc/mcMc.c
node/nodeWffPrint.c
parser/input.l
parser/symbols.h
parser/grammar.y
prop/propDb.c
sm/smMain.c
```

4 CHANGELOG (details)

File name	**Changes**
compile/compileCone.c	line 1592 onwards: introduced support for EAX, EEA, AAA (universal quantifier), AAX, EAF, EAG, AAF, AAG
compile/compileFlatten.c	line 1592 onwards: introduced support for EAX, EEA, AAA, AAX, EAF, EAG, AAF, AAG
mc/mc.h	added definitions of new functions eax, eau, aau, eaf, eag (line 73-)
mc/mcEval.c	New `case:` statements for EAX, EAU, AAU, AAX, EAF, EAF, AAF, AAG (AAX, AAF, AAG defined in terms of EAX, EAG, EAF)
mc/mcMc.c	New functions eax, eau, aau, eaf, eag
node/nodeWffPrint.c	Using arity 10 and 11 to print new operators nicely
parser/input.l	Added tokens: TOK_EAX, TOK_EAU, TOK_AAA, TOK_AAX, TOK_EAF, TOK_EAG, TOK_AAF, TOK_AAG
parser/grammar.y	Added rules for TOK_EAX, TOK_EAU, TOK_AAA, TOK_AAX, TOK_EAF, TOK_EAG, TOK_AAF, TOK_AAG
parser/symbols.h	Added new symbols EAX, EAU, AAU, AAX, EAF, EAG, AAF, AAG
prop/propDb.c	Disabled check for IVAR's (line 242). **THIS MUST BE REINTRODUCED!**
sm/smMain.c	BannerPrint changed

5 Examples

I created a directory /arctl-tests under /nusmv, with the following examples:

5.1 Example 1

```
-- File: post-projection.smv
-- See p.6 Charles notes
MODULE main
  VAR state : {a,b,c,d};
  IVAR action : {0,1};

  INIT ( state=a )

  TRANS ( next(state) = case
              state=a : b;
              (state=b & action=0): c;
              (state=b & action=1): d;
              1: state;
           esac
         )
```

```
SPEC AG(state=b -> (EX(state=c) & EX(state=d)))
-- More interesting: doing action=1 from b always leads to d
SPEC AG(state=b -> !(EAX(action=1)(!state=d)))
```

5.2 Example 2

```
-- File: latch.smv
MODULE main
  VAR output: boolean;
  IVAR input: boolean;
       clock: boolean;
  ASSIGN
    init(output) := 0;
    next(output) := case
        clock = 0 : output;
        clock = 1 : input;
     esac;

-- Test
SPEC EX(output=1)
-- If clock and input, then output
SPEC AG(!EAX(clock=1 & input=1)(!(output=1)))
-- False: there is no way to have output=0 if clock & input
SPEC EF(EAX(clock=1 & input=1)(output=0))
-- If clock, then output (false, could be input=0)
SPEC AG(!EAX(clock=1)(!(output=1)))
-- This is true everywhere
SPEC TRUE -> AAX(clock=1 & input=1)(output=1)
```

5.3 Example 3

```
-- File:
MODULE agent
  IVAR move: boolean;
  VAR count : 0..10;
  ASSIGN
    init(count) :=0 ;
    next(count) := case
                     move & count < 10: count + 1;
                     1 : count;
                  esac;
  DEFINE win := (count=10);

MODULE main
  VAR alice : agent;
  VAR bob : agent;

SPEC !(EAX(bob.move)(bob.count=0))
SPEC EAX(!bob.move)(bob.count>0)
SPEC EA(!alice.move)[TRUE U alice.win]
SPEC !(EA(!alice.move)[TRUE U alice.win])
SPEC AAX(bob.move & alice.move)(bob.count > 0 & alice.count > 0)
SPEC !AAX(bob.move)(bob.count > 0)
SPEC AA(bob.move)[!alice.win U bob.win]
SPEC AA(bob.move)[!bob.win U alice.win]
SPEC EAF(bob.move)(alice.win)
SPEC EAF(bob.move)(!alice.win)
SPEC EAF(alice.move)(!alice.win)
SPEC EAF(!alice.move)(alice.win)
SPEC AAF(alice.move)(alice.win)
SPEC EAG(bob.move)(!alice.win)
SPEC AAF(alice.move)(bob.win)
SPEC AAG(bob.move)(bob.count>0)
SPEC START -> EF(bob.win) & EF(!bob.win)
SPEC !START -> (EF(bob.win))
```

6 Open issues

- Check the correctness of the code.
- Introduce check on expressions with IVAR (see above).
- Generation of traces?