CS 267: Automated Verification

Lecture 7: SMV Symbolic Model Checker, Partitioned Transition Systems, Counter-example Generation in Symbolic Model Checking

Instructor: Tevfik Bultan
SMV [McMillan 93]

- BDD-based symbolic model checker
- Finite state
- Temporal logic: CTL
- Focus: hardware verification
  - Later applied to software specifications, protocols, etc.
- SMV has its own input specification language
  - concurrency: synchronous, asynchronous
  - shared variables
  - boolean and enumerated variables
  - bounded integer variables (binary encoding)
    - SMV is not efficient for integers, but that can be fixed
  - fixed size arrays
SMV Language

• An SMV specification consists of a set of modules (one of them must be called main)
• Modules can have access to shared variables
• Modules can be composed asynchronously using the process keyword
• Module behaviors can be specified using the ASSIGN statement which assigns values to next values of variables in parallel
• Module behaviors can also be specified using the TRANS statements which allow specification of the transition relation as a logic formula where next state values are identified using the next keyword
Two concurrently executing processes are trying to enter a critical section without violating mutual exclusion.
Example Mutual Exclusion Protocol in SMV

MODULE process1(a,b,turn)
VAR
   pc: {out, wait, cs};
ASSIGN
   init(pc) := out;
   next(pc) :=
      case
         pc=out : wait;
         pc=wait & (!b | !turn) : cs;
         pc=cs : out;
         1 : pc;
      esac;
next(turn) :=
   case
      pc=out : 1;
      1 : turn;
   esac;
next(a) :=
   case
      pc=out : 1;
      pc=cs : 0;
      1 : a;
   esac;
next(b) := b;
FAIRNESS
running

MODULE process2(a,b,turn)
VAR
   pc: {out, wait, cs};
ASSIGN
   init(pc) := out;
   next(pc) :=
      case
         pc=out : wait;
         pc=wait & (!a | turn) : cs;
         pc=cs : out;
         1 : pc;
      esac;
next(turn) :=
   case
      pc=out : 0;
      1 : turn;
   esac;
next(b) :=
   case
      pc=out : 1;
      pc=cs : 0;
      1 : b;
   esac;
next(a) := a;
FAIRNESS
running
Example Mutual Exclusion Protocol in SMV

```plaintext
MODULE main
VAR
    a : boolean;
    b : boolean;
    turn : boolean;
    p1 : process process1(a,b,turn);
    p2 : process process2(a,b,turn);
SPEC
    AG(!(p1.pc=cs & p2.pc=cs))
    -- AG(p1.pc=wait -> AF(p1.pc=cs)) & AG(p2.pc=wait -> AF(p2.pc=cs))
```

Here is the output when I run SMV on this example to check the mutual exclusion property

```plaintext
% smv mutex.smv
-- specification AG (!(p1.pc = cs & p2.pc = cs)) is true
resources used:
user time: 0.01 s, system time: 0 s
BDD nodes allocated: 692
Bytes allocated: 1245184
BDD nodes representing transition relation: 143 + 6
```
Example Mutual Exclusion Protocol in SMV

The output for the starvation freedom property:

```
% smv mutex.smv
-- specification AG (p1.pc = wait -> AF p1.pc = cs) & AG ... is true

resources used:
user time: 0 s, system time: 0 s
BDD nodes allocated: 1251
Bytes allocated: 1245184
BDD nodes representing transition relation: 143 + 6
```
Example Mutual Exclusion Protocol in SMV

Let’s insert an error

change \[ pc=\text{wait} \& (\neg b \mid \neg \text{turn}) : cs; \]

to \[ pc=\text{wait} \& (\neg b \mid \text{turn}) : cs; \]
-- specification AG (!(p1.pc = cs & p2.pc = cs)) is false
-- as demonstrated by the following execution sequence

state 1.1:
a = 0
b = 0
turn = 0
p1.pc = out
p2.pc = out
[stuttering]

state 1.2:
[executing process p2]

state 1.3:
b = 1
p2.pc = wait
[executing process p2]

state 1.4:
p2.pc = cs
[executing process p1]

state 1.5:
a = 1
turn = 1
p1.pc = wait
[executing process p1]

state 1.6:
p1.pc = cs
[stuttering]

resources used:
user time: 0.01 s, system time: 0 s
BDD nodes allocated: 1878
Bytes allocated: 1245184
BDD nodes representing transition relation: 143 + 6
Symbolic Model Checking with BDDs

• As we discussed earlier BDDs are used as a data structure for encoding trust sets of Boolean logic formulas in symbolic model checking

• One can use BDD-based symbolic model checking for any finite state system using a Boolean encoding of the state space and the transition relation

• Why are we using symbolic model checking?
  – We hope that the symbolic representations will be more compact than the explicit state representation on the average
  – In the worst case we may not gain anything
Symbolic Model Checking with BDDs

- Possible problems
  - The BDD for the transition relation could be huge
    - Remember that the BDD could be exponential in the number of disjuncts and conjuncts
    - Since we are using a Boolean encoding there could be a large number of conjuncts and disjuncts
  - The EX computation could result in exponential blow-up
    - Exponential in the number of existentially quantified variables
Partitioned Transition Systems

- If the BDD for the transition relation $R$ is too big, we can try to partition it and represent it with multiple BDDs.
- We need to be able to do the EX computation on this partitioned transition system.
Disjunctive Partitioning

- Disjunctive partitioning:
  \[ R \equiv R_1 \lor R_2 \lor \ldots \lor R_k \]

  We can distribute the EX computation since existential quantification distributes over disjunction

  We compute the EX for each \( R_i \) separately and then take the disjunction of all the results
Disjunctive Partitioning

- Remember EX, let’s assume that EX also takes the transition relation as input:
  \[ EX(p, R) = \{ s | (s,s') \in R \text{ and } s' \in p \} \]
  which in symbolic model checking becomes:
  \[ EX(p, R) \equiv \exists V' \ R \land p[V' / V] \]

If we can write R as \( R \equiv R_1 \lor R_2 \lor \ldots \lor R_k \) then
\[
EX(p, R) \equiv \exists V' \ R \land p[V' / V]
\equiv \exists V' \ (R_1 \lor R_2 \lor \ldots \lor R_k) \land p[V' / V]
\equiv \exists V' \ (R_1 \land p[V' / V] \lor R_2 \land p[V' / V] \lor \ldots \lor R_k \land p[V' / V])
\equiv (\exists V' \ R_1 \land p[V' / V]) \lor (\exists V' \ R_2 \land p[V' / V]) \lor \ldots \lor (\exists V' \ R_k \land p[V' / V])
\equiv EX(p, R_1) \lor EX(p, R_2) \lor \ldots \lor EX(p, R_k)
Disjunctive Partitioning

The purpose of disjunctive partitioning is the following:

• If we can write $R$ as
  \[ R \equiv R_1 \lor R_2 \lor \ldots \lor R_k \]
  then we can use $R_1 \ldots R_k$ instead of $R$ during the EX computation and we never have to construct the BDD for $R$

• We can use $R_i$s to compute the EX($p, R$) as
  \[ \text{EX}(p, R) \equiv \text{EX}(p, R_1) \lor \text{EX}(p, R_2) \lor \ldots \lor \text{EX}(p, R_k) \]

• If $R$ is much bigger than all the $R_i$s, then disjunctive partitioning can improve the model checking performance
Recall this Extremely Simple Example

Variables: \( x, y \): boolean
Set of states:
\( S = \{(F,F), (F,T), (T,F), (T,T)\} \)
\( S \equiv \text{True} \)

Initial condition:
\( I \equiv \neg x \land \neg y \)

Transition relation (negates one variable at a time):
\( R \equiv x' = \neg x \land y' = y \lor x' = x \land y' = \neg y \) \quad (= \text{means } \leftrightarrow) \)

A possible disjunctive partitioning:
\( R \equiv R_1 \lor R_2 \)
\( R_1 \equiv x' = \neg x \land y' = y \quad R_2 \equiv x' = x \land y' = \neg y \)
An Extremely Simple Example

Given \( p = x \land y \), compute \( \text{EX}(p) \)

\[
\text{EX}(p, R) \equiv \exists V' \ R \land p[V' / V]
\]
\[
\equiv \text{EX}(p, R_1) \lor \text{EX}(p, R_2)
\]

\[
\text{EX}(p, R_1) \equiv (\exists V' \ R_1 \land x' \land y') \equiv (\exists V' \ x' = \neg x \land y' = y \land x' \land y')
\]
\[
\equiv (\exists V' \ x \lor y \land x' \land y') \equiv \neg x \land y
\]

\[
\text{EX}(p, R_2) \equiv (\exists V' \ R_2 \land x' \land y') \equiv (\exists V' \ x' = x \land y' = x \land y' \land y')
\]
\[
\equiv (\exists V' \ x \land \neg y \land x' \land y') \equiv x \land \neg y
\]

\( \text{EX}(x \land y) \equiv \text{EX}(p, R_1) \lor \text{EX}(p, R_2) \equiv \neg x \land y \lor x \land \neg y \)

In other words \( \text{EX}({(T,T)}) \equiv \{(F,T), (T,F)\} \)
Conjunctive Partitioning

• Conjunctive partitioning:
  \[ R \equiv R_1 \land R_2 \land \ldots \land R_k \]

  Unfortunately EX computation does not distribute over the conjunction partitioning in general since \textit{existential quantification does NOT distribute over conjunction}

• However if each \( R_i \) is expressed on a separate set of next state variables (i.e., if a next state variable appears in \( R_i \) then it should not appear in any other conjunct)
  – Then we can distribute the existential quantification over each \( R_i \)
**Conjunctive Partitioning**

- If we can write $R$ as $R \equiv R_1 \land R_2 \land \ldots \land R_k$
  where $R_i$ is a formula only on variables $V_i$ and $V_i'$
  and $i \neq j \Rightarrow V_i' \cap V_j' = \emptyset$
  which means that a primed variable does not appear in more than one $R_i$

- Then, we can do the existential quantification separately for each $R_i$ as follows:
  \[
  \text{EX}(p, R) \equiv \exists V' R \land p[V' / V] \\
  \equiv \exists V' p[V' / V] \land (R_1 \land R_2 \land \ldots \land R_k) \\
  \equiv (\exists V_k' \ldots (\exists V_2' (\exists V_1' p[V' / V] \land R_1) \land R_2) \ldots \land R_k)
  \]
An Even Simpler Example

Variables: $x, y$: boolean
Set of states:
$S = \{(F,F), (F,T), (T,F), (T,T)\}$
$S \equiv \text{True}$

Initial condition:
$I \equiv \neg x \land \neg y$

Transition relation (negates one variable at a time):
$R \equiv x' = \neg x \land y' = \neg y$  
($= \text{means } \leftrightarrow$)

A possible conjunctive partitioning:
$R \equiv R_1 \land R_2$
$R_1 \equiv x' = \neg x$  
$R_2 \equiv y' = \neg y$
An Even Simpler Example

Given \( p \equiv x \land y \), compute \( \text{EX}(p) \)

\[
\text{EX}(p, R) \equiv \exists V' \ R \land p[V' / V]
\]
\[
\equiv \exists V_2' \ (\exists V_1' \ p[V' / V] \land R_1) \land R_2
\]
\[
\equiv \exists V_2' \ (\exists V_1' \ x' \land y' \land R_1) \land R_2
\]
\[
\equiv \exists y' \ (\exists x' \ x' \land y' \land x' = \neg x) \land y' = \neg y
\]
\[
\equiv \exists y' \ (\exists x' \ x' \land y' \land \neg x) \land y' = \neg y
\]
\[
\equiv \exists y' \ y' \land \neg x \land y' = \neg y
\]
\[
\equiv \exists y' \ y' \land \neg x \land \neg y
\]
\[
\equiv \neg x \land \neg y
\]

\[
\text{EX}(x \land y) \equiv \neg x \land \neg y
\]

In other words \( \text{EX}((T,T)) \equiv \{(F,F)\} \)
Partitioned Transition Systems

- Using partitioned transition systems we can reduce the size of memory required for representing $R$ and the size of the memory required to do model checking with $R$

- Note that, for either type of partitioning
  - disjunctive $R \equiv R_1 \lor R_2 \lor \ldots \lor R_k$
  - or conjunctive $R \equiv R_1 \land R_2 \land \ldots \land R_k$

  size of $R$ can be exponential in $k$

- So by keeping $R$ in partitioned form we can avoid constructing the BDD for $R$ which can be exponentially bigger than each $R_i$
Other Improvements for BDDs

• Variable ordering is important
  – For example for representing linear arithmetic constraints such as \( x = y + z \) where \( x, y, \) and \( z \) are integer variables represented in binary,
  • If the variable ordering is: all the bits for \( x \), all the bits for \( y \) and all the bits for \( z \), then the size of the BDD is exponential in the number of bits
    – In fact this is the ordering used in SMV which makes SMV very inefficient for verification of specifications that contain arithmetic constraints
  • If the binary variables for \( x, y, \) and \( z \) are interleaved, the size of the BDD is linear in the number of bits
    – So, for specific classes of systems there may be good variable orderings
Other Improvements to BDDs

• There are also dynamic variable ordering heuristics which try to change the ordering of the BDD on the fly and reduce the size of the BDD

• There are also variants of BDDs such as multi-terminal decision diagrams, where the leaf nodes have more than two distinct values.
  – Useful for domains with more than two values
    • Can be translated to BDDs
Counter-Example Generation

• Remember: Given a transition system $T = (S, I, R)$ and a
  CTL property $p$ $T \models p$ iff for all initial state $s \in I$, $s \models p$

• Verification vs. Falsification
  – Verification:
    • Show: initial states $\subseteq$ truth set of $p$
  – Falsification:
    • Find: a state $\in$ initial states $\cap$ truth set of $\neg p$
    • Generate a counter-example starting from that state

• The ability to find counter-examples is one of the biggest
  strengths of the model checkers
An Example

• If we wish to check the property AG(p)

• We can use the equivalence:
  $AG(p) \equiv \neg EF(\neg p)$

If we can find an initial state which satisfies $EF(\neg p)$, then we know that the transition system $T$, does not satisfy the property $AG(p)$
Another Example

• If we wish to check the property $\text{AF}(p)$

• We can use the equivalence:
  
  $$\text{AF}(p) \equiv \neg \text{EG}(\neg p)$$

If we can find an initial state which satisfies $\text{EG}(\neg p)$, then we know that the transition system $T$, does not satisfy the property $\text{AF}(p)$
General Idea

- We can define two temporal logics using subsets of CTL operators
  - ACTL: CTL formulas which only use the temporal operators AX, AG, AF and AU and all the negations appear only in atomic properties (there are no negations outside of temporal operators)
  - ECTL: CTL formulas which only use the temporal operators EX, EG, EF and EU and all the negations appear only in atomic properties

- Given an ACTL property its negation is an ECTL property
Counter-Example Generation

• Given an ACTL property $p$, we negate it and compute the set of states which satisfy it is negation $\neg p$
  – $\neg p$ is an ECTL property

• If we can find an initial state which satisfies $\neg p$ then we generate a counter-example path for $p$ starting from that initial state
  – Such a path is called a *witness* for the ECTL property $\neg p$
An Example

- We want to check the property $AG(p)$
- We compute the fixpoint for $EF(\neg p)$
- We check if the intersection of the set of initial states $I$ and the truth set of $EF(\neg p)$ is empty
  - If it is not empty we generate a counter-example path starting from the intersection

$EF(\neg p) \equiv$ states that can reach $\neg p \equiv \neg p \cup EX(\neg p) \cup EX(EX(\neg p)) \cup ...$

- In order to generate the counter-example path, save the fixpoint iterations.
- After the fixpoint computation converges, do a second pass to generate the counter-example path.

Generate a counter-example path starting from a state here
Another Example

- We want to check the property $AF(p)$
- We compute the fixpoint for $EG(\neg p)$
- We check if the intersection of the set of initial states $I$ and the truth set of $EG(\neg p)$ is empty
  - If it is not empty we generate a counter-example path starting from the intersection

$EG(\neg p) \equiv \text{states that can avoid reaching } p \equiv \neg p \cap EX(\neg p) \cap EX(EX(\neg p)) \cap \ldots$

- In order to generate the counter-example path, look for a cycle in the resulting fixpoint

Generate a counter-example path starting from a state here
Counter-example generation

• In general the counter-example for an ACTL property (equivalently a witness to an ECTL property) is not a single path
• For example, the counter example for the property $\text{AF}(\text{AG}p)$ would be a witness for the property $\text{EG}(\text{EF}\neg p)$
  – It is not possible to characterize the witness for $\text{EG}(\text{EF}\neg p)$ as a single path
• However it is possible to generate tree-like transition graphs containing counter-example behaviors as a counter-example:
  – Edmund M. Clarke, Somesh Jha, Yuan Lu, Helmut Veith: “Tree-Like Counterexamples in Model Checking”. LICS 2002: 19-29