

Formal Specification and Verification

Temporal logic, part 3

26.06.2012

Viorica Sofronie-Stokkermans
e-mail: sofronie@uni-koblenz.de

Temporal logic

Linear Time Logic

Syntax: Π set of propositional variables.

The set of LTL formulae is the smallest set such that:

- \top, \perp and each propositional variable $P \in \Pi$ are formulae;
- if F, G are formulae, then so are $F \wedge G, F \vee G, \neg F$;
- if F, G are formulae, then so are $\bigcirc F$ and $F \mathcal{U} G$

Semantics: Transition systems; computations π

Temporal logic

Branching Time Logic: CTL

Syntax: Π set of propositional variables.

The class of CTL formulae is the smallest set such that

- \top, \perp and each propositional variable $P \in \Pi$ are formulae;
- if F, G are formulae, then so are $F \wedge G, F \vee G, \neg F$;
- if F, G are formulae, then so are
 $A \circ F$ and $E \circ F$,
 $A(F \mathcal{U} G)$ and $E(F \mathcal{U} G)$.

The symbols A and E are called path quantifiers.

We also use:

$A \diamond F$ for $A(\top \mathcal{U} F)$

$E \diamond F$ for $E(\top \mathcal{U} F)$

$A \square F$ for $\neg E \diamond \neg F$

$E \square F$ for $\neg A \diamond \neg F$

CTL: Semantics

Let $T = (S, \rightarrow, L)$ be a transition system. We define satisfaction of CTL formulas in T at states $s \in S$ as follows:

$(T, s) \models p$	iff	$p \in L(s)$
$(T, s) \models \neg F$	iff	$(T, s) \models F$ is not the case
$(T, s) \models F \wedge G$	iff	$(T, s) \models F$ and $(T, s) \models G$
$(T, s) \models F \vee G$	iff	$(T, s) \models F$ or $(T, s) \models G$
$(T, s) \models E \circ F$	iff	$(T, t) \models F$ for some $t \in S$ with $s \rightarrow t$
$(T, s) \models A \circ F$	iff	$(T, t) \models F$ for all $t \in S$ with $s \rightarrow t$
$(T, s) \models A(FUG)$	iff	for all computations $\pi = s_0 s_1 \dots$ of T with $s_0 = s$, there is an $m \geq 0$ such that $(T, s_m) \models G$ and $(T, s_k) \models F$ for all $k < m$
$(T, s) \models E(FUG)$	iff	there exists a computation $\pi = s_0 s_1 \dots$ of T with $s_0 = s$, such that there is an $m \geq 0$ such that $(T, s_m) \models G$ and $(T, s_k) \models F$ for all $k < m$

Model Checking

The CTL model checking problem is as follows:

Given a transition system $T = (S, \rightarrow, L)$ and a CTL formula F , check whether T satisfies F , i.e., whether $(T, s) \models F$ for all $s \in S$.

Method (Idea)

- (1) Arrange all subformulas F_i of F in a sequence F_0, \dots, F_k in ascending order w.r.t. formula length: for $1 \leq i < j \leq k$, F_i is not longer than F_j ;
- (2) For all subformulas F_i of F , compute the set

$$\text{sat}(F_i) := \{s \in S \mid (T, s) \models F_i\}$$

in this order (from shorter to longer formulae);

- (3) Check whether $S \subseteq \text{sat}(F)$.

Model Checking

How to compute $sat(F_i)$

- $p \in \Pi \mapsto sat(p) = \{s \mid L(p, s) = 1\}$
- $sat(F_i \wedge F_j) = sat(F_i) \cap sat(F_j)$
- $sat(\neg F_i) = S \setminus sat(F_i)$
- $sat(E \circ F_i) = \{s \mid \exists t \in S : (s \rightarrow t) \wedge t \in sat(F_i)\}$
- $sat(A \circ F_i) = \{s \mid \forall t \in S : (s \rightarrow t) \wedge t \in sat(F_i)\}$
- $sat(E(F_i \mathcal{U} F_j))$ and $sat(A(F_i \mathcal{U} F_j))$ are computed with the following procedures:

Model Checking

$$F = E(F_i \cup F_j)$$

```
sat(F) := T := sat(F_j)
while T != {} do
  choose s in T
  T := T \ {s}
  for all t in S with t -> s do
    if t in sat(F_i) and t not in sat(F) then
      sat(F) := sat(F) U {t}
      T := T U {t}
```

$$F = A(F_i \cup F_j)$$

```
sat(F) := T := sat(F_j)
while T != {} do
  choose s in T
  T := T \ {s}
  for all t in S with t -> s do
    flag = 1
    for all t' in S with t -> t' do
      if t' not in sat(F) then flag := 0
    if t in sat(F_i) and t not in sat(F) and flag = 1 then
      sat(F) := sat(F) U {t}
      T := T U {t}
```

Model Checking

Theorem. $(T, s) \models F$ iff $s \in \text{sat}(F)$.

Consequence. CTL model checking is decidable.

Concerning the complexity, we observe the following: if F is of length n , then at most n sets $\text{sat}(F_i)$ need to be computed. How complex is it to compute each such set?

- F is a propositional letter or of the form $F_1 \wedge F_2$ or $\neg F_1$: $O(|S|)$ steps needed;
- F is of the form $E \circ F_i$ or $E(F_1 \mathcal{U} F_2)$: $O(|S| + |\rightarrow|)$ steps needed
the maximum cardinality of the initial set $\text{sat}(F_j)$ is $|S|$, and, in the forall loop, each edge from \rightarrow is “touched” at most once (in all iterations of the while);
- F is of the form $A(F_1 \mathcal{U} F_2)$: $O(|S| + |\rightarrow|^2)$ steps needed
the maximum cardinality of the initial set $\text{sat}(F_j)$ is $|S|$, the outer forall loop touches each edge from \rightarrow at most once, and the inner forall loop touches each edge at most once for each step done by the outer forall loop.

There exist more efficient algorithms (complexity $|F| \cdot O(|S| + |\rightarrow|)$).

Model Checking

Theorem. $(T, s) \models F$ iff $s \in \text{sat}(F)$.

Idea of the proof: Structural induction, taking into account that:

- $\text{sat}(\top) = S$, $\text{sat}(\perp) = \emptyset$, $\text{sat}(p) = \{s \mid p \in L(s)\}$, $p \in \Pi$
- $\text{sat}(\neg F) = S \setminus \text{sat}(F)$; $\text{sat}(F \wedge G) = \text{sat}(F) \cap \text{sat}(G)$
- $\text{sat}(E \circ F) = \{s \in S \mid \text{Post}(s) \wedge \text{sat}(F) \neq \emptyset\}$
- $E(FUG) \equiv G \vee (F \wedge E \circ E(FUG))$

$\text{Sat}(E(FUG))$ is the smallest subset T of S such that

- (1) $\text{sat}(F) \subseteq T$ (2) $s \in \text{sat}(F)$ and $\text{Post}(s) \cap T \neq \emptyset$ implies $s \in T$

- $E\Box F \equiv F \wedge E \circ E\Box F$

$\text{sat}(E\Box F)$ is the largest subset T of S such that:

- (1) $T \subseteq \text{sat}(F)$ (2) $s \in T$ implies $\text{Post}(s) \cap T \neq \emptyset$

- $\text{sat}(A(FUG))$ is the smallest subset T of S satisfying

$$\text{sat}(G) \cup \{s \in \text{sat}(F) \mid \text{Post}(s) \subseteq T\} \subseteq T$$

Model checking

Sufficient to have method for computing

$sat(E \circ F)$, $sat(E(FUG))$ and ($sat(E\Box F)$ or $sat(A\Diamond F)$)

All other formulae starting with path quantifiers can be expressed in terms of $E \circ F$, $E(FUG)$ and $E\Box F$ or $A\Diamond$.

- $A \circ F \equiv \neg E \circ \neg F$
- $A(FUG) \equiv \neg E(\neg GU\neg F \wedge \neg G) \wedge \neg E\Box\neg G$
- $A\Diamond F \equiv \neg E\neg\Diamond F \equiv \neg E\Box\neg F$
- ...

Algorithm

```
function SAT( $F$ ) /* determines the set of states satisfying  $F$  */ begin
  case
     $F = \top$ : return  $S$ 
     $F = \perp$ : return  $\emptyset$ 
     $F$  is atomic: return  $\{s \in S \mid F \in L(s)\}$ 
     $F = \neg G$ : return  $S - SAT(G)$ 
     $F = G_1 \wedge G_2$ : return  $SAT(G_1) \cap SAT(G_2)$ 
     $F = G_1 \vee G_2$ : return  $SAT(G_1) \cup SAT(G_2)$ 
     $F = A \circ F$ : return  $SAT(\neg E \circ \neg F)$ 
     $F = E \circ F$ : return  $SAT_{E \circ}(F)$ 
     $F = A(FUG)$ : return  $SAT(\neg E(\neg GU(\neg F \wedge \neg G)) \wedge \neg E \square \neg G)$ 
     $F = E(FUG)$ : return  $SAT_{EU}(F, G)$ 
     $F = E \diamond F$ : return  $SAT(E(\top U F))$ 
     $F = E \square F$ : return  $SAT(\neg A \diamond \neg F)$ 
     $F = A \diamond F$ : return  $SAT_{A \diamond}(F)$ 

     $F = A \square F$ : return  $SAT(\neg E(\top U \neg F))$ 
```

Algorithm

The algorithm and its subfunctions use program variables X , Y , V and W which are sets of states.

The program for SAT handles the easy cases directly and passes more complicated cases on to special procedures, which in turn might call SAT recursively on subexpressions.

These special procedures rely on implementations of the functions

$$pre_{\exists}(Y) = \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in Y)\}$$

$$pre_{\forall}(Y) = \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in Y)\}.$$

Algorithm

function $SAT_{E\circ}(F)$

begin

$X := SAT(F)$

$Y := pre_{\exists}(X)$

 return Y

end

Algorithm

function $SAT_{EU}(F_i, F_j)$

```
sat(F) := T := sat(F_j)
while T  $\neq$  {} do
  choose s in T
  T := T \ {s}
  for all t in S with t  $\rightarrow$  s do
    if t in sat(F_i) and t not in sat(F) then
      sat(F) := sat(F) U {t}
      T := T U {t}
```

Algorithm

function $SAT_{EU}(F_i, F_j)$

```
sat(F) := T := sat(F_j)
while T  $\neq$  {} do
  choose s in T
  T := T \ {s}
  for all t in S with t  $\rightarrow$  s do
    if t in sat(F_i) and t not in sat(F) then
      sat(F) := sat(F) U {t}
      T := T U {t}
```

function $SAT_{EU}(F, G)$

```
begin
  W = SAT(F)
  X = S
  Y = SAT(G)
  repeat until X = Y
  begin
    X = Y;
    Y := Y U (W  $\cap$   $pre_{\exists}$ (Y))
  end
  return Y
end
```

Algorithm

function $SAT_{A\Diamond}(F)$

```
sat(A\Diamond F) := T := sat(F)
while T  $\neq$  {} do
  choose s in T
  T := T \ {s}
  for all t in S with t  $\rightarrow$  s do
    flag = 1
    for all t' in S with t  $\rightarrow$  t' do
      if t' not in sat(A\Diamond F) then flag := 0
    if t not in sat(A\Diamond F) and flag = 1 then
      sat(A\Diamond F) := sat(A\Diamond F) U {t}
      T := T U {t}
```

function $SAT_{A\Diamond}(F)$

```
begin
  X := S
  Y := SAT(F)
  repeat un til X = Y
  begin
    X := Y
    Y := Y  $\cup$   $pre_{\forall}(Y)$ 
  end
  return Y
end
```

The state explosion problem

Although the algorithm is linear in the size of the model, unfortunately the size of the model is itself more often than not exponential in the number of variables and the number of components of the system which execute in parallel.

This means that, for example, adding a boolean variable to your program will double the complexity of verifying a property of it.

The tendency of state spaces to become very large is known as the state explosion problem. A lot of research has gone into finding ways of overcoming it, including, e.g. the use of:

- Efficient data structures, called ordered binary decision diagrams (OBDDs), which represent sets of states instead of individual states.
- Abstraction: one may interpret a model abstractly, uniformly or for a specific property.

OBDDs

We start by showing how sets of states are represented with OBDDs, together with some of the operations required.

Then, we extend that to the representation of the transition system.

Finally, we show how the remainder of the required operations is implemented.

Representing subsets of the set of states

Let S be a finite set (we forget for the moment that it is a set of states).

The task is to represent the various subsets of S as OBDDs.

Since OBDDs encode boolean functions, we need somehow to code the elements of S as boolean values.

The way to do this in general is to assign to each element $s \in S$ a unique vector of boolean values (v_1, v_2, \dots, v_n) , each $v_i \in \{0, 1\}$.

Then, we represent a subset T by the boolean function f_T which maps (v_1, v_2, \dots, v_n) onto 1 if $s \in T$ and maps it onto 0 otherwise.

Representing subsets of the set of states

In the case that S is the set of states of a transition system $T = (S, \rightarrow, L)$, there is a natural way of choosing the representation of S as boolean vectors.

The labelling function $L : S \rightarrow \mathcal{P}(\Pi)$ gives us the encoding. (Fix ordering on the atoms in Π , say p_1, \dots, p_n)

$$s \in S \mapsto (v_1, \dots, v_n) \in \{0, 1\}^n, \text{ where } v_i = \begin{cases} 1 & \text{if } p_i \in L(s) \\ 0 & \text{if } p_i \notin L(s) \end{cases}$$

As an OBDD, this state is represented by the OBDD of the boolean function $l_1 \wedge l_2 \wedge \dots \wedge l_n$, where l_i is p_i if $p_i \in L(s)$ and $\neg p_i$ otherwise.

Representing subsets of the set of states

The set of states $\{s_1, s_2, \dots, s_m\}$ is represented by the OBDD of the boolean function

$$(l_{11} \wedge l_{12} \wedge \dots \wedge l_{1n}) \vee (l_{21} \wedge l_{22} \wedge \dots \wedge l_{2n}) \vee \dots \vee (l_{m1} \wedge l_{m2} \wedge \dots \wedge l_{mn})$$

where $l_{i1} \wedge l_{i2} \wedge \dots \wedge l_{in}$ represents state s_i .

OBDDs

In order to justify the claim that the representation of subsets of S as OBDDs will be suitable for the algorithm presented before,

We need to look at how the operations on subsets which are used in that algorithm can be implemented in terms of the operations we have defined on OBDDs. The operations in that algorithm are:

- (1) Intersection, union and complementation of subsets.

It is clear that these are represented by the boolean functions \wedge , \vee and \neg respectively.

OBDDs

In order to justify the claim that the representation of subsets of S as OBDDs will be suitable for the algorithm presented before,

We need to look at how the operations on subsets which are used in that algorithm can be implemented in terms of the operations we have defined on OBDDs. The operations in that algorithm are:

(2) The functions

$$pre_{\exists}(X) = \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in X)\}$$

$$pre_{\forall}(X) = \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in X)\}.$$

The function pre_{\exists} takes a subset X of states and returns the set of states which can make a transition into X . The function pre_{\forall} takes a set X and returns the set of states which can make a transition only into X .

OBDDs

In order to justify the claim that the representation of subsets of S as OBDDs will be suitable for the algorithm presented before,

We need to look at how the operations on subsets which are used in that algorithm can be implemented in terms of the operations we have defined on OBDDs. The operations in that algorithm are:

(1) Intersection, union and complementation of subsets.

(2) The functions

$$pre_{\exists}(X) = \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in X)\}$$

$$pre_{\forall}(X) = \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in X)\}.$$

In order to see how $pre_{\exists}(X)$, $pre_{\forall}(X)$ are implemented in terms of OBDDs, we need first to look at how the transition relation itself is represented.

Representing the transition relation

The transition relation \rightarrow of a model $T = (S, \rightarrow, L)$ is a subset of $S \times S$.

We have already seen that subsets of a given finite set may be represented as OBDDs by considering the characteristic function of a binary encoding.

Just like in the case of subsets of S , the binary encoding is naturally given by the labelling function L . Since \rightarrow is a subset of $S \times S$, we need two copies of the boolean vectors.

Representing the transition relation

The transition relation \rightarrow of a model $T = (S, \rightarrow, L)$ is a subset of $S \times S$.

We have already seen that subsets of a given finite set may be represented as OBDDs by considering the characteristic function of a binary encoding.

Just like in the case of subsets of S , the binary encoding is naturally given by the labelling function L . Since \rightarrow is a subset of $S \times S$, we need two copies of the boolean vectors.

Representing the transition relation

Thus, the link $s \rightarrow s'$ is represented by the pair of Boolean vectors $((v_1, \dots, v_n), (v'_1, \dots, v'_n))$, where

- $v_i = 1$ iff $p_i \in L(s)$ and
- $v'_i = 1$ iff $p_i \in L(s')$.

As an OBDD, the link is represented by the OBDD for the boolean function

$$(l_1 \wedge l_2 \wedge \dots \wedge l_n) \wedge (l'_1 \wedge l'_2 \wedge \dots \wedge l'_n)$$

and a set of links (for example, the entire relation \rightarrow) is the OBDD for the \vee of such formulas.

Implementing the functions pre_{\exists} and pre_{\forall}

It remains to show how an OBDD for $pre_{\exists}(X)$ and $pre_{\forall}(X)$ can be computed, given OBDDs B_X for X and B_{\rightarrow} for the transition relation \rightarrow .

First, we observe that pre_{\forall} can be expressed in terms of complementation and pre_{\exists} , as follows:

$$pre_{\forall}(X) = S \setminus pre_{\exists}(S \setminus X),$$

where $S \setminus Y = \{s \in S \mid s \notin Y\}$.

Therefore, we need only explain how to compute the OBDD for $pre_{\exists}(X)$ in terms of B_X and B_{\rightarrow} .

Implementing the functions pre_{\exists} and pre_{\forall}

We proceed as follows:

1. Rename the variables in B_X to their primed versions; call the resulting OBDD $B_{X'}$.
2. Compute the OBDD for $exists(\hat{p}', apply(\wedge, B_X, B_{X'}))$ using the apply and exists algorithms for OBDDs.

Synthesising OBDDs

It might be too time consuming to compute the OBDD for the transition relation by first computing the truth table and then an OBDD which might not be in its fully reduced form (and hence needs to be reduced).

The key idea and attraction of applying OBDDs to finite systems is therefore to take a system description in a **specialized language** and to synthesise the OBDD directly, without having to go via intermediate representations (such as binary decision trees or truth tables) which are exponential in size.

The specialized languages should allow us to define the next values of the variables in terms of their current values – compiled into a set of boolean functions f_1, \dots, f_n , where f_i defines the next value of p_i in terms of the current values of all the variables.

The boolean function representing the transition relation is therefore of the form

$$\bigwedge_{i=1}^n p'_i \leftrightarrow f_i$$

Overview

- **Model checking:**

Finite transition systems / CTL properties

States are “entities” (no precise description, except for labelling functions)

No precise description of actions (only \rightarrow important)

Overview

- **Model checking:**

Finite transition systems / CTL properties

States are “entities” (no precise description, except for labelling functions)

No precise description of actions (only \rightarrow important)

Extensions in two possible directions:

- More precise description of the actions/events
 - Propositional Dynamic Logic
 - Hoare logic
- More precise description of states (and possibly also of actions)
 - succinct representation: formulae represent a set of states
 - deductive verification