Formal Specification and Verification

Temporal logic (Part 5)

24.01.2017

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

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 satifies F, i.e., whether $(T, s) \models F$ for all $s \in S$.

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 satifies 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 $F_0, \ldots F_k$ 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

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

in this order (from shorter to longer formulae);

(3) Check whether $S \subseteq sat(F)$.

Model Checking

How to compute $sat(F_i)$

- $p \in \Pi \mapsto sat(p) = \{s \mid L(p, s) = 1\}$
- $sat(\neg F) = S \backslash sat(F)$
- $sat(F \wedge G) = sat(F) \cap sat(G)$
- $sat(F \vee G) = sat(F) \cup sat(G)$
- $sat(E \bigcirc F) = \{s \mid \exists t \in S : (s \to t) \wedge t \in sat(F)\}$
- $sat(A \bigcirc F) = \{s \mid \forall t \in S : (s \to t) \text{ implies } t \in sat(F)\}$
- $sat(E(F\mathcal{U}G))$ and $sat(A(F\mathcal{U}G)$ are computed as explained in what follows.

Model Checking

Lemma. $sat(E(F\mathcal{U}G))$ is the smallest set T with

(1) $sat(G) \subseteq T$

(2) $s \in sat(F)$ and $\mathrm{Post}(s) \cap T \neq \emptyset$ implies $s \in T$

Remark: $E(F\mathcal{U}G)$ is a fixpoint of the equation $\Phi \equiv G \vee (F \wedge E \bigcirc \Phi)$.

$sat(E(F\mathcal{U}G))$ can be computed iteratively as follows:

$$T_0 := sat(G)$$

$$T_{i+1} := T_i \cup \{s \in sat(F) \mid Post(s) \cap T_i \neq \emptyset\}$$

Then: $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_j \subseteq T_{j+1} \subseteq \cdots \subseteq sat(E(F\mathcal{U}G))$.

Since S is finite, there exists j such that $T_j = T_{j+1} = \ldots$.

This T_j will be $sat(E(F\mathcal{U}G))$.

Model checking

Remark: $sat(A\Diamond F)$ is the smallest set T with

(1) $sat(F) \subseteq T$

(2) $\{s \in S \mid \forall s'(s \to s' \text{ implies } s' \in T)\} \subseteq T$.

It can be computed iteratively as follows:

$$T_0 := sat(F)$$

$$T_{i+1} := T_i \cup \{s \in S \mid \forall s'(s \to s' \text{ implies } s' \in T_i)\} = T_i \cup \text{pre}_{\forall}(T_i)$$

Then: $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_j \subseteq T_{j+1} \subseteq \cdots \subseteq sat(A\Diamond F)$.

Since S is finite, there exists j such that $T_j = T_{j+1} = \ldots$.

This T_j will be $sat(A\Diamond F)$.

Model checking

Remark: $sat(E\Box F)$ is the largest set T with

(1) $T \subseteq sat(F)$

(2) $s \in T$ implies $\text{Post}(s) \cap T \neq \emptyset$.

It can be computed iteratively as follows:

$$T_0 := sat(F)$$

$$T_{i+1} := T_i \cap \{s \in sat(F) \mid Post(s) \cap T_i \neq \emptyset\}$$

Then: $T_0 \supseteq T_1 \supseteq \cdots \supseteq T_j \supseteq T_{j+1} \supseteq \cdots \supseteq sat(E\Box F)$.

Since S is finite, there exists j such that $T_j = T_{j+1} = \ldots$.

This T_j will be $sat(E\Box F)$.

Model checking

Sufficient to have method for computing

$sat(E \bigcirc F), sat(E(F\mathcal{U}G))$ and $(sat(E\Box F)$ or $sat(A\Diamond F))$

All other formulae starting with path quantifiers can be expressed in terms of $E \bigcirc F$, $E(F\mathcal{U}G)$ and $E\Box F$ or $A\Diamond$.

$$
\begin{aligned}
A \bigcirc F &\equiv \neg E \bigcirc \neg F \\
A(F\mathcal{U}G) &\equiv \neg E(\neg G\mathcal{U}\neg F \wedge \neg G)) \wedge \neg E\Box\neg G \\
A\Diamond F &\equiv \neg E\neg\Diamond F \equiv \neg E\Box\neg F \\
&\vdots
\end{aligned}
$$

Algorithm

function SAT(F) /* determines the set of states satisfying F */

begin

case

$F = \top$:	return S
$F = \bot$:	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 $SAT(G_1) \cup SAT(G_2)$
$F = A \bigcirc F$:	return $SAT(\neg E \bigcirc \neg F)$
$F = E \bigcirc F$:	return $SAT_{E\bigcirc}(F)$
$F = A(F \mathcal{U} G)$:	return $SAT(\neg E(\neg G \mathcal{U}(\neg F \wedge \neg G)) \wedge \neg E \Box \neg G)$
$F = E(F \mathcal{U} G)$:	return $SAT_{EU}(F, G)$
$F = E \Diamond F$:	return $SAT(E(\top \mathcal{U} F))$
$F = E \Box F$:	return $SAT(\neg A \Diamond \neg F)$
$F = A \Diamond F$:	return $SAT_{A\Diamond}(F)$
$F = A \Box F$:	return $SAT(\neg E(\top \mathcal{U} \neg F))$

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

$pre_{\exists}(Y) = \{s \in S |$ exists $s', (s \to s'$ and $s' \in Y)\}$

$pre_{\forall}(Y) = \{s \in S |$ for all $s', (s \to s'$ implies $s' \in Y)\}$.

Algorithm

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

begin

X := SAT(F)

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

return Y

end

Algorithm

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

```
begin
   W := SAT(F)
   X := S
   Y := SAT(G)
   repeat until X = Y
   begin
      X := Y
      Y := Y ∪ (W ∩ pre∃(Y))
   end
   return Y
end
```

Algorithm

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

```
begin
    X := S
    Y := SAT(F)
    repeat until X = Y
    begin
        X := Y
        Y := Y ∪ pre∀(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.

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 $(v_1, v_2, ..., v_n)$, each $v_i \in 0, 1$.

Then, we represent a subset T by the boolean function f_T which maps $(v_1, v_2, ..., v_n)$ 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.

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 $p_1, \ldots, p_n$)

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

As an OBDD, this state is represented by the OBDD of the boolean function $l_1 \wedge l_2 \wedge \cdots \wedge l_n$, 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 $\{s_1, s_2, \ldots, s_m\}$ is represented by the OBDD of the boolean function

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

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

Examples

Example (.pdf file) linked separately.

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:

(1) Intersection, union and complementation of subsets.

(2) The functions

$pre_{\exists}(X) = \{s \in S |$ exists $s', (s \to s'$ and $s' \in X)\}$

$pre_{\forall}(X) = \{s \in S |$ for all $s', (s \to s'$ 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

$$pre_{\exists}(X) = \{s \in S| \text{ exists } s', (s \to s' \text{ and } s' \in X)\}$$
$$pre_{\forall}(X) = \{s \in S| \text{ for all } s', (s \to s' \text{ implies } s' \in X)\}.$$

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

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

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

and a set of links (for example, the entire relation $\to$) 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 $B_\to$ for the transition relation $\to$.

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

$$pre_\forall(X) = S \backslash pre_\exists(S \backslash X),$$

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

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

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 $exists(\overline{p}', apply(\wedge, B_\rightarrow, B_{X'}))$ 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 $f_1, \ldots, f_n$, 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