SAPIENZA Università di Roma

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2011/2012

Course: Elective in Software and Services

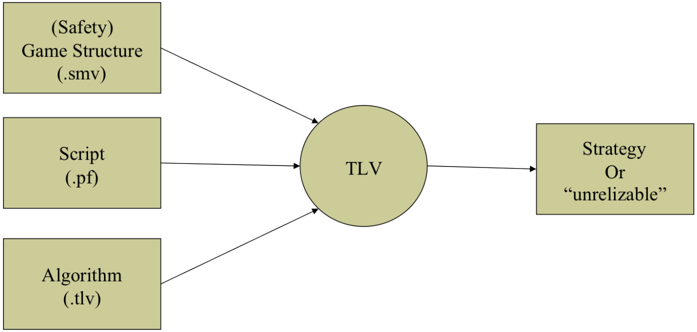
Solving Safety-Games with TLV

Fabio Patrizi

`patrizi@dis.uniroma1.it`

May 2012

Solving the Game



- Assume the input Safety-Game structure (i.e., the SMV-file) is given
- We want to build an algorithm for TLV that:
  - computes the winning set of the game
  - returns all the winning strategies for the system (if any)

TLV-Basic

TLV-Basic: the TLV language for functions and algorithms

Programming language that provides:

- Access to the game structure:
  - States
  - Transitions
- Functions for creation, deletion, and manipulation of states and transitions

The `.tlv` and `.pf` files are written in this language

Example

Recall module `main`:

```
MODULE main
VAR
  env: system Env(sys.index); -- sys #1 is environment
  sys: system Sys; -- sys #2 is system
DEFINE
  good := ...
```

This is a TLV-Basic fragment:

```
-- this is a comment
To prepare_synt; -- function declaration
  Let vars1 := V(1); --variables of sys #1
  Let vars2 := V(2); --variables of sys #2
  Let prv1 := prime(V(1)); --"primed" variables of sys #1
  Let prv2 := prime(V(2)); --"primed" variables of sys #2
  Let trans1 := Ti(1,1); -- the transition relation of sys #1
  Let trans2 := Ti(1,2); -- the transition relation of sys #2
End -- To prepare_synt;
```

Variables

2 classes of variables:

- *System* variables: those defined in the game structure (.smv file)
  - we have *primed* and *unprimed* versions
- *Dynamic* variables: those manipulated by TLV-BASIC

TLV-BASIC programs can only read system variables

Note: System variables can range over finite sets only (recall SMV files). W.l.o.g. we assume they are always boolean!

TLV-BASIC programs create dynamic variables when assigning values through the instruction `Let` (which defines a *global variable*), e.g.,:

```
Let prv1 := prime(V(1));
```

To delete a variable, use `delvar` instead, e.g.: `delvar prv1;`

Data Types

TLV-Basic essentially provides the following data structures:

- integer values;
- propositional formulas;
- sets of system variables.

(In fact, they are internally represented as *Ordered Binary Decision Diagrams* (OBDD), but we don't see the details.)
NOTE: Formulas are not evaluated!

Expressions

TLV-Basic expressions are built from constants, variables and the following syntactic elements:

- Logical operators: `|` (OR), `&` (AND), `!` (NOT), `->` (IMPLICATION), `->` (IFF);
- Numeric operators: `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `mod`;
- application of functions `prime()` and `next()`;
- *variable quantification* (see forthcoming slides):
  - `expr forsome expr2`;
  - `expr forall expr2`;
- Other (not considered here).

For instance, assuming `a`, `c`, and `d` are variables:

```
a | (c & next(d));
```

is a (formula) expression.

Semantics of Propositional Formulas

Given a propositional formula, its *semantics* is the set of its propositional models.
E.g., the formula $a|b$ (a OR b), where a and b are system variables, has 3 models:

- $a = 1$ (or `true`), $b = 0$ (or `false`);
- $a = 0,\ b = 1$;
- $a = 1,\ b = 1$.

Formulas as Sets of States

Thus, we can use formulas to isolate states of a transition system with (system) variables $\mathrm{v}_1, \dots, \mathrm{v}_n$.

For instance, if the transition system has variables $\mathrm{v}_1, \dots, \mathrm{v}_4$, the formula v_1 & $!\mathrm{v}_3$ identifies all the states s.t. $\mathrm{v}_1 = 1$ and $\mathrm{v}_3 = 0$, i.e.:

- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 0,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 0$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 0,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 1$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 1,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 0$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 1,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 1$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 0,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 0$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 0,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 1$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 1,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 0$
- $\mathrm{v}_1 = 1,\ \mathrm{v}_2 = 1,\ \mathrm{v}_3 = 0,\ \mathrm{v}_4 = 1$

Formulas as Sets of Transitions

By using *primed* versions of a transition system variable, we can isolate transitions of a transition system with (system) variables $\mathtt{v}_1, \ldots, \mathtt{v}_n$.

We adopt the convention that primed variables stand for values at successor states

For instance, if the transition system has variables $\mathtt{v}_1, \mathtt{v}_2$, the formula $\mathtt{v}_1$ & $(\mathtt{v}_1' \mid \mathtt{v}_2')$ identifies the following transitions:

- $\langle 1, 0 \rangle \rightarrow \langle 0, 1 \rangle$
- $\langle 1, 0 \rangle \rightarrow \langle 1, 0 \rangle$
- $\langle 1, 0 \rangle \rightarrow \langle 1, 1 \rangle$
- $\langle 1, 1 \rangle \rightarrow \langle 0, 1 \rangle$
- $\langle 1, 1 \rangle \rightarrow \langle 1, 0 \rangle$
- $\langle 1, 1 \rangle \rightarrow \langle 1, 1 \rangle$
- ...

Formulas as Sets of Transitions (cont.)

The transition relation of a game structure in TLV is internally represented as a propositional formula possibly containing primed system variables

The transition relation of the environment can be accessed with the invocation `Ti(1,1)`, e.g.,

```
Let trans1 := Ti(1,1);
```

The transition relation of the system can be accessed with the invocation `Ti(1,2)`, e.g.,

```
Let trans2 := Ti(1,2);
```

Thus, the transition relation of the whole game structure is:

```
Let trans := trans1 & trans 2;
```

Formulas as Sets of Transitions (cont.)

In fact, the transition relation of a module is defined as a formula, e.g.:

```
TRANS
  case
    state = start_st & op = start_op : next(state) = t0 & next(op) in {a};
    state = t0  & op = a : next(state) = t1 & next(op) in {b};
    state = t1  & op = b  : next(state) = t2 & next(op) in {b,c};
    state = t2  & op = b : next(state) = t2 & next(op) in {b,c};
    state = t2  & op = c: next(state) = t0 & next (op) in {a};
  esac
```

`next(v)` identifies the *primed version* of a variable `v`

Alternatively, the more general `prime()` can be used, which can be applied to a whole expression, and turns all of its variables into their primed version

Variable Quantification

Variable quantification provides a compact way to represent conjunctions and disjunctions of formulas.

Assume $\mathtt{V}$ is a dynamic variable assigned to a set of system variables, e.g., $\mathtt{V} = \{\mathtt{v}'_1, \mathtt{v}'_2\}$.

The expression

$$\mathtt{v}_1 \ \& \ (\mathtt{v}'_1 \mid \mathtt{v}'_2) \ \ \textit{forsome}\ \mathtt{V}$$

Is equivalent to:

$(\mathtt{v}_1 \ \& \ (\mathtt{v}'_1 = 0 \mid \mathtt{v}'_2 = 0))|(\mathtt{v}_1 \ \& \ (\mathtt{v}'_1 = 1 \mid \mathtt{v}'_2 = 0))|$
$(\mathtt{v}_1 \ \& \ (\mathtt{v}'_1 = 0 \mid \mathtt{v}'_2 = 1))|(\mathtt{v}_1 \ \& \ (\mathtt{v}'_1 = 1 \mid \mathtt{v}'_2 = 1))$

Observe in this formula only $\mathtt{v}_1$ occurs as a free variable.

Variable Quantification (cont.)

$\mathtt{V} = \{\mathtt{v}_1', \mathtt{v}_2'\}$

The expression

$$\mathtt{v}_1 \ \& \ (\mathtt{v}_1' \mid \mathtt{v}_2') \ \ \textit{forall}\ \mathtt{V}$$

Is equivalent to:
$(\mathtt{v}_1 \ \& \ (\mathtt{v}_1' = 0 \mid \mathtt{v}_2' = 0))\&(\mathtt{v}_1 \ \& \ (\mathtt{v}_1' = 1 \mid \mathtt{v}_2' = 0))\&$
$(\mathtt{v}_1 \ \& \ (\mathtt{v}_1' = 0 \mid \mathtt{v}_2' = 1))\&(\mathtt{v}_1 \ \& \ (\mathtt{v}_1' = 1 \mid \mathtt{v}_2' = 1))$

Observe in this formula only $\mathtt{v}_1$ occurs as a free variable.

$\odot$ Operator

- The $\odot$ operator is the basic building block of the μ-calc on GSs
- It is the atomic step in any (fixpoint) computation of the controller's winning states
- To actually compute winning states, we need to implement $\odot$
- This can be done by suitably combining the constructs seen so far

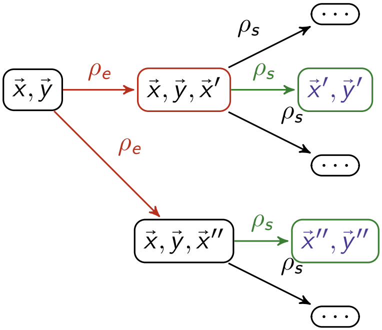
The $\odot$ Operator

Recall that $\langle \vec{x}, \vec{y} \rangle \models \odot \Psi$ iff

$$\exists \vec{x}'.\rho_e(\vec{x}, \vec{y}, \vec{x}')$$

$$\wedge$$

$$\forall \vec{x}'.\rho_e(\vec{x}, \vec{y}, \vec{x}') \rightarrow \exists \vec{y}'.\rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \ s.t. \ \langle \vec{x}', \vec{y}' \rangle \models \Psi$$



The ⊙ Operator: TLV encoding

```
...
--Remember: Let defines global variables!
Let vars1 := V(1);
Let vars2 := V(2);
Let prv1 := prime(V(1));
Let prv2 := prime(V(2));
Let trans1 := Ti(1,1);
Let trans2 := Ti(1,2);
...
Func cox(psi); --function definition
  Let exy := ((trans2 & prime(psi)) forsome prv2);
  Let rcox := ((trans1 -> exy) forall prv1);
  Return rcox;
End -- end of Func cox(psi);
```

The $\odot$ Operator: TLV encoding (cont.)

- `trans2` is the TLV encoding for ρ_s. Thus,

  ```
  ((trans2 & prime(psi)) forsome prv2)
  ```

  is the TLV encoding for

  $$\exists \vec{y}'.\rho_s(\vec{x},\vec{y},\vec{x}',\vec{y}')\ s.t.\ \langle \vec{x}',\vec{y}' \rangle \vDash \Psi$$

- `trans1` is the TLV encoding for ρ_e. Thus,

  ```
  ((trans1 -> exy) forall prv1)
  ```

  is the TLV encoding for

  $$\forall \vec{x}'.\rho_e(\vec{x},\vec{y},\vec{x}') \rightarrow \exists \vec{y}'.\rho_s(\vec{x},\vec{y},\vec{x}',\vec{y}')\ s.t.\ \langle \vec{x}',\vec{y}' \rangle \vDash \Psi$$

  which is the expression that `cox` returns.

The missing conjunct

$$\exists \vec{x}'.\rho_e(\vec{x},\vec{y},\vec{x}')$$

is not required as we assume that every reachable state has a successor.

Winning Set Computation

We are now ready to compute the winning set for the system:

- this is the set of states that satisfy the formula

$$\boxdot\varphi = \nu X.\varphi \wedge \odot X$$

- because the system is finite-state, this can be computed by:
  - iteratively applying the $\odot$ operator
  - starting from the goal φ
  - terminating when a fixpoint is reached (guaranteed in a finite sequence of steps)

Winning Set Computation (cont.)

TLV provides a convenient construct for fixpoints:

```
Fix(exp) S End
```

This repeats the body `S` until the evaluation of `exp` does not change

`Fix(z)` is equivalent to the more intuitive:

```
While (new_temp != exp)
  Let new_temp := exp;
  S
End
```

Winning Set Computation (cont.)

Thus, function `winm` defined below computes the winning set for the system, for invariant `invp`

```
Func winm(invp);
  Let z := 1;
  Fix (z)
    Let z := invp & cox(z);
  End -- Fix (z)
  Return z;
End -- Func winm(invp);
```

Is the Game Winning for the system?

Once we have computed the winning set for the system, we can check whether the game is winning for the system

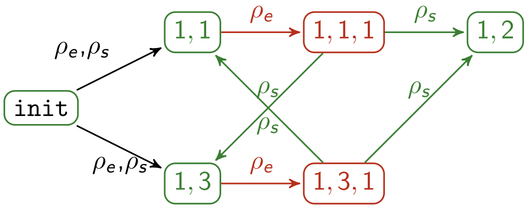
To do so, we need to check that all initial states are winning for the system

In TLV, this can be easily done as follows:

```
...
Let win := winm(good); --good is the formula defined in main
Let init := I(1) & I(2); --env and sys initial formulas
Let winning_for_sys := init -> win; -- all initial states are winning
...
```

Computing the System Strategy

In general, the winning set is not sufficient to compute a system strategy. This is the case of, e.g., $\diamondsuit$good, for: `good := x = 1 & y = 2`



We may not know the strategy!

In these cases, one needs to keep intermediate information as the fixpoint computation progresses in *winm*()

In the special case of safety games, however, the winning set is a finite representation of all possible strategies (e.g., $\boxdot$`(x=1)`).

Computing the System Strategy (cont.)

As an example, the following computes a set of loop-free strategies for $\boxdot \diamond$ good:

```
Let trans12 := trans1 & trans2;

Let z := good;
Let trans := 0;

-- compute <>good
Fix(z)
  Let new_z := cox(z);
  --Store the transitions witnessing the existence of the strategy:
  Let trans := trans | (new_z & !z & trans12 & next(z));
  Let z := z | new_z;
End

--compute []<>good
Fix(z)
  Let z := z & cox(z);
End

Let trans := trans | (I(1) & I(2) & trans12 & prime(z & !(I(1) & I(2))) |
     (goal & z & trans12 & next(z));
```

The involved files

Typically, an algorithm in SMV-Basic involves 2 files:

- a `.tlv` file where the basic functions are defined (see above)
- a `.pf` file (also referred to as the *script*) where the algorithm is built by invoking functions defined in the file above

The script file

An example of script file

```
Load "safety-game.tlv"; -- loads the functions defined in safety-game.tlv
To do_synthesis; -- define function do_synthesis
  check_realizability; -- check whether the game is winning for the system
  If (realizable)
    symb_strategy; -- compute the strategies (OG)
    check_symb; -- check that all states have a successor
    -- print the OG:
    enumerate_symb;
    print_automat;
  End -- If (realizable)
End -- To do_synthesis;

do_synthesis; --invoke function
quit;
```

All functions used but not defined in this file are defined in `safety-game.tlv`

In principle, everything could be put in the script file

However, in practice many functions defined in the `.tlv` file are general-purpose

Running the script file

To run the script, execute the following command:

```
tlv script.pf gs.tlv
```

where:

- `script.pf` is a valid TLV-Basic script;
- `gs.tlv` is an SMV file where a game structure is encoded in a module `main` containing, the environment and the system specification, plus the definition of the invariant formula `good`.

Further References

Further information about TLV:

- TLV home page: `http://www.cs.nyu.edu/acsys/tlv/`
- TLV manual (downloadable from the home page)