stv_v2 Manual

Institute of Computer Science of Polish Academy of Sciences

September 6, 2024

Contents

1 Usage

Basic functionality of stv_v2.

1.1 Build

```
cd build
make clean
make
```

or

```
cd build
./build-run
```

1.2 Run

```
./stv
```

2 Changing default settings

How to change default run of stv_v2.

2.1 Config file

Some options can be changed straight from the config file that is in:

```
build/config.txt
```

2.2 Options

The following options are acceptable:

- Change model file path:

```
./stv --file PATH_TO_MODEL
./stv -f PATH_TO_MODEL
```

- Change mode:
  - Normal mode:

```
./stv -m 0
```

  - Generate GlobalModel:

```
./stv -m 1
```

  - Run verification:

```
./stv -m 2
```

  - Generate GlobalModel and run verification:

```
./stv -m 3
```

2.3 Flags

Avaliable flags:

- stdout data on global model (after expandAllStates):

```
--OUTPUT_GLOBAL_MODEL
```

- stdout data on local models:

```
--OUTPUT_LOCAL_MODELS
```

- generate .dot files for agent templates, local and global models:

```
--OUTPUT_DOT_FILES
```

- generate global models with epsilon transitions:

```
--ADD_EPSILON_TRANSITIONS
```

- replace the formula from the model file with a different one:

```
--OVERWRITE_FORMULA [formula]
```

```
--OVERWRITE_FORMULA "FORMULA: ↩
   <<Voter1>><>(Voter1_vote == 1)"
```

Example formula replacement with another one.

- output counterexample path if the formula verification returns FALSE:

```
--COUNTEREXAMPLE
```

- reduce the amount of states and transitions using a DFS-POR algorithm and select the first correct transition:

```
--REDUCE [agents]
```

```
--REDUCE ""
```

Example reduction with no specified agent.

```
--REDUCE "Agent1 Agent2"
```

Example reduction with 2 specified agents.

- reduce the amount of states and transitions using a DFS-POR algorithm and select all avaliable transitions:

```
--REDUCE_ALL [agents]
```

```
--REDUCE_ALL ""
```

Example reduction with no specified agent.

```
--REDUCE_ALL "Agent1 Agent2"
```

Example reduction with 2 specified agents.

3 Creating a model

Model creation info.

3.1 Model description

3.1.1 Agents

Describing a new agent starts with a new line with a keyword *Agent* followed by the name of the agent and a colon.

```
Agent newAgentName:
```

Creating an agent with a name "newAgentName".

Then a few agent description lines follow.

- In a new line starting with a keyword $LOCAL$, we add local variable names in square brackets, separated by a by a comma if there's more than one variable. We can also leave the brackets empty.

  ```
  LOCAL: [variableName1, variableName2, ...]
  ```

  Creating local variables named "variableName1", "variableName2".

- In a new line starting with a keyword $PERSISTENT$, we add variable names of the variables, which value should be remembered at all times. As previously, we add the names in square brackets, separated by a

comma if there's more than one variable. We can also leave the brackets empty.

```
PERSISTENT: [variableName1, variableName2, ...]
```

Making local variables named "variableName1", "variableName2" and so on, persistent.

- In a new line starting with a keyword $INITIAL$, we can set the values of the created variables using an := operator after the variable name and assigning it an integer. As previously, we add the names in square brackets, separated by a comma if there's more than one variable. We can also leave the brackets empty. The default value for each variable is 0.

  ```
  INITIAL: [variableName1:=7777, ...]
  ```

  Setting a value 7777 to the "variableName1" variable and so on.

- In a new line starting with a keyword *init*, we have to set the name of the initial local state from which the agent is going to start.

  ```
  init q0
  ```

  Setting the "q0" as the initial state in the local model.

3.1.2 Actions

Following the previous lines, we can specify the actions avaliable for the agent, each new action in a new line. First we enter the action name, then after a colon we specify the avaliable action by writing a state, from which the model will be able to execute the action, then an "arrow" $(->)$ operator and finally a state to which we will move after executing the action.

```
action1: q0 -> q1
action2: q1 -> q2
action3: q2 -> q0
```

Making a loop out of 3 actions: action1, action2 and action3.

3.1.3 Shared actions

If the action will be shared with another agent he have to start the action by writing a keyword *shared*. Then in square brackets we have to write and integer which will specify how many agents will use this action, current agent included. Then, after writing a shared action name, we write a local action name in square brackets. If the local action name will be the same as the shared name, it means that the current agent will be the one performing a synchronization. Otherwise, agent can only wait for the synchronization to use the shared action. After that, the action specification continues as usual.

```
shared[2] action1[action1]: q0 -> q1
```

Creating an action that will synchronize with 1 other agent. The current agent will be the one performing the sync.

```
shared[3] action1[someAction]: q0 -> q1
```

Creating an action that will synchronize with 2 other agents. The current agent will be the one waiting for sync.

3.1.4 Conditions

To add a condition for an action to fire, we have to add the condition between the square brackets between the initial state name and the arrow operator.

```
action1: q0 [someLocalValue==1] -> q1
```

Local action that can only fire if someLocalValue is equal to 1.

3.1.5 Variable change

To change the variable after the action fires, we have to add the change between the square brackets after the name of the state that we moved to in this action. If you have to assign multiple variables, you can separate the variable assignment instances with a comma.

```
action1: q0 -> q1 [someLocalValue:=1, localValue:=2]
```

Action that sets the value of someLocalValue to 1 and localValue to 2.

```
action1: q0 -> q1 [someLocalValue:=7777]
```

Action that sets the value of someLocalValue to 7777.

3.1.6 Comments

To add a comment, put a # in a text line. Every alphanumeric character, including spaces and tabs from this point on will be treated as a comment, until a new line begins.

```
action1: q0 -> q1 #this is a comment
action2: q0 -> q1 # this is also a comment
action3: q0 -> q1 #   this   is   a   comment   too
# this also works
```

Example comments in a file.

3.2 Formula description

Formula description starts in a new line with a keyword $FORMULA$. After a colon, you enter the optional agent coalition in double angle brackets, for which you want to verify the formula. If you omit specifying the agent coalition, the formula will be verified for CTL instead. Then, you use one of the symbols used for verification:

- `[]` – G – Globally
- `<>` – F – Finally

Additionally you can specify the following knowledge operators:

- $\&K\_agent(formula)$ – Knowledge (Agent name needed after the "_". Lastly, you write the formula that you want to check in round brackets.)
- $\&H\_agent[<= k/ >= k]((fromula1), (fromula2), ...)$ – Hartley entropy (Agent name needed after the "_". Then a <= or >= followed by an integer are required, wrapped in square brackets. Lastly, you write the formulas that you want to check in round brackets. Allows for listing more than one formula, separated with commas, each formula encapsulated within round brackets.)

```
FORMULA: <<Agent1>>[](value==1 || value==7777)
```

A formula that checks if there is a strategy that globally Agent1 could take such actions that the value will be always equal to 1 or 7777.

```
FORMULA: <<Agent1>><>K_Agent2(value==1 || value==7777)
```

A formula that uses a knowledge operator.

```
FORMULA: <<Agent1>>[]H_Agent2[>=3]((value==1 || ↩
   value==7777),(value==2))
```

A formula that uses a Hartley operator.

3.3 Avaliable operators

List of avaliable operators for conditions, variable changes and formula description:

- `:=` – Variable assignment.
- `||` – Logic OR.
- `&&` – Logic AND.
- `==` – Equal.
- `!=` – Not equal.
- `>=` – Greater or equal.
- `>` – Greater.
- `<=` – Less or equal.
- `<` – Less.
- `()` – Brackets.
- `!` – Negation of a variable.

- + – Addition of two variables.
- - – Subtraction of two variables.
- * – Multiplication of two variables.
- / – Division of two variables, returns an integer.
- % – Modulo of two variables.
- $\&K\_agent(formula)$ – Knowledge
- $\&H\_agent[<= k/ >= k]((fromula1), (fromula2), ...)$ – Hartley entropy

3.4 Running a specific model

As mentioned previously, use the corresponding option to change the model file path.

3.5 Example model

A classic example model of two voters trying to vote as they want, choosing between two candidates. There's also a coercer, trying to force them to vote on a specific candidate by punishing them or not after casting a vote.

```
Agent Voter1:
LOCAL: [Voter1_vote]
PERSISTENT: [Voter1_vote]
INITIAL: []
init q0
voter1vote1: q0 -> q1 [Voter1_vote:=1]
shared[2] gv_1_Voter1[gv_1_Voter1]: q1 ↩
   [Voter1_vote==1] -> q2
voter1vote2: q0 -> q1 [Voter1_vote:=2]
shared[2] gv_2_Voter1[gv_2_Voter1]: q1 ↩
   [Voter1_vote==2] -> q2
shared[2] ng_Voter1[ng_Voter1]: q1 -> q2
shared[2] pun_Voter1[pn_Voter1]: q2 -> q3
shared[2] npun_Voter1[pn_Voter1]: q2 -> q3
idle: q3->q3
```

```
Agent Voter2:
LOCAL: [Voter2_vote]
PERSISTENT: [Voter2_vote]
INITIAL: []
init q0
voter2vote1: q0 -> q1 [Voter2_vote:=1]
shared[2] gv_1_Voter2[gv_1_Voter2]: q1 ↩
   [Voter2_vote==1] -> q2
voter2vote2: q0 -> q1 [Voter2_vote:=2]
shared[2] gv_2_Voter2[gv_2_Voter2]: q1 ↩
   [Voter2_vote==2] -> q2
shared[2] ng_Voter2[ng_Voter2]: q1 -> q2
shared[2] pun_Voter2[pn_Voter2]: q2 -> q3
shared[2] npun_Voter2[pn_Voter2]: q2 -> q3
idle: q3->q3

Agent Coercer1:
LOCAL: [Coercer1_Voter1_vote, Coercer1_Voter1_gv, ↩
   Coercer1_pun1, Coercer1_npun1, ↩
   Coercer1_Voter2_vote, Coercer1_Voter2_gv, ↩
   Coercer1_pun2, Coercer1_npun2]
PERSISTENT: [Coercer1_Voter1_vote, ↩
   Coercer1_Voter1_gv, Coercer1_pun1, Coercer1_npun1, ↩
   Coercer1_Voter2_vote, Coercer1_Voter2_gv, ↩
   Coercer1_pun2, Coercer1_npun]
INITIAL: []
init q0
shared[2] gv_1_Voter1[g_Voter1]: q0 -> q1 ↩
   [Coercer1_Voter1_vote:=1, Coercer1_Voter1_gv:=1]
shared[2] gv_2_Voter1[g_Voter1]: q0 -> q1 ↩
   [Coercer1_Voter1_vote:=2, Coercer1_Voter1_gv:=1]
shared[2] ng_Voter1[g_Voter1]: q0 -> q1 ↩
   [Coercer1_Voter1_gv:=2]
shared[2] pun_Voter1[pun_Voter1]: q2 -> q3 ↩
   [Coercer1_pun1:=1]
shared[2] npun_Voter1[npun_Voter1]: q2 -> q3 ↩
   [Coercer1_npun1:=1]

shared[2] gv_1_Voter2[g_Voter2]: q1 -> q2 ↩
```

```
   [Coercer1_Voter2_vote:=1, Coercer1_Voter2_gv:=1]
shared[2] gv_2_Voter2[g_Voter2]: q1 -> q2 ↩
   [Coercer1_Voter2_vote:=2, Coercer1_Voter2_gv:=1]
shared[2] ng_Voter2[g_Voter2]: q1 -> q2 ↩
   [Coercer1_Voter2_gv:=2]
shared[2] pun_Voter2[pun_Voter2]: q3 -> q4 ↩
   [Coercer1_pun2:=1]
shared[2] npun_Voter2[npun_Voter2]: q3 -> q4 ↩
   [Coercer1_npun2:=1]

FORMULA: <<Voter1>>[](((Voter1_vote == 1) && ↩
   (Coercer1_npun1 == 1)) || ((Voter2_vote == 1) && ↩
   (Coercer1_npun2 == 1)))
```