SRML Code Syntax Guide

September 10, 2018

Systems in EVE are modelled with the *Simple Reactive Modules Language* (SRML), that can be used to model non-deterministic systems. Each system component (agent/player) in SRML is represented as a SRML *module*. We also associate each module (except *environment module*) with a goal, which is specified as an LTL formula. For E-Nash and A-Nash, a property to be checked (specified as an LTL formula) is also required.

General Structure of SRML Program

An SRML program is composed of two sections: *modules declarations* and *property declaration*. The latter is optional for Non-Emptiness problem, but compulsory for E-Nash and A-Nash. Using EVE , one can model both RMG and CGS. The first one is played in an arena implicitly given by the specification of the players in the game (as done in [2]), the second one is played on a graph, *e.g.*, as done in [1]. So in general, an SRML program structure is as follows:

<table>
<tr><th>RMG</th><th>CGS</th></tr>
<tr><td><code><module></code>
...
<code><module></code>
<code><property></code></td><td><code><module></code>
...
<code><module></code>
<code><environment_module></code>
<code><property></code></td></tr>
</table>

Formal declarations of modules and property are shown below.

1. **Modules declaration.** An agent is represented as a *module*, which consists of an **interface** that defines the name of the module and lists a non-empty set of Boolean variables controlled by the module, and a set of **guarded commands**,

which define the choices available to the module at each state. There are two kinds of guarded commands: **init**, used for initialising the variables, and **update**, used for updating variables subsequently. An associated LTL goal for each module is also declared.

A <module> is declared as follows:

```
module <agentID> controls <controlled_vars>
 init
 :: true ~> <action>;
...
 update
 :: <guard> ~> <action>;
...
 goal
 :: <LTL_formula>;
```

where

- <agentID> is any string starting with a letter and followed by any number of letters, digits, or underscore sign. Formally,

  <agentID> :: [a-zA-Z][a-zA-Z0-9_]*

- <controlled_vars> is the set of controlled variables, separated by commas. Formally

  <controlled_vars> :: <varID>","...","<varID>

  where <varID> is any string starting with a letter and followed by any number of letters, digits, or underscore sign: <varID> :: [a-zA-Z][a-zA-Z0-9_]*

- <guard> is a propositional logic formula built inductively from the set of Boolean variables with the following syntax:

Propositional Formula	SRML Syntax
$\top$	true
$\bot$	false
$\neg\varphi$	! *formula*
$\varphi \wedge \varphi$	*formula* and *formula*
$\varphi \vee \varphi$	*formula* or *formula*
$\varphi \rightarrow \varphi$	*formula* -> *formula*
$\varphi \leftrightarrow \varphi$	*formula* <-> *formula*

Table 1: Propositional formula to SRML syntax translation.

- <action> defines how to update the value of the controlled variables. It is in the following form

```
<action> ::
  <varID>"'":="<prop_formula>","...","<varID>"'":="<prop_formula>
```

  where <prop_formula> is built inductively using the same syntax shown in Table 1.

- <LTL_formula> is built inductively by using the following grammar, where $p \in \Phi$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

  The translation from LTL formula to SRML syntax is shown in the following table:

LTL Formula	SRML Syntax
$\top$	`true`
$\bot$	`false`
$\neg\varphi$	`!` *formula*
$\varphi \wedge \varphi$	*formula* `and` *formula*
$\varphi \vee \varphi$	*formula* `or` *formula*
$\varphi \rightarrow \varphi$	*formula* `->` *formula*
$\varphi \leftrightarrow \varphi$	*formula* `<->` *formula*
$\mathsf{X}\,\varphi$	`X` *formula*
$\mathsf{F}\,\varphi$	`F` *formula*
$\mathsf{G}\,\varphi$	`G` *formula*
$\varphi\,\mathsf{U}\,\varphi$	*formula* `U` *formula*

Table 2: LTL formula to SRML syntax translation.

To make it clearer, please consider an example below, the right side column is the SRML code tranlsation of the left side column.

module m_a **controls** p
  init
  $:: \top \rightsquigarrow p' := \top;$
  update
  $:: p \vee q \rightsquigarrow p' := \top;$
  $:: q \rightsquigarrow p' := \neg p;$
  goal
  $:: \mathsf{G}\,\mathsf{F}\,p;$

```
module ma controls p
  init
  ::  true ~> p':= true;
  update
  ::  p or q ~> p':= true;
  ::  q ~> p':= !p;
  goal
  ::  G F p;
```

2. **Environment module declaration.** Environment module is declared similarly to the (normal) module, except that `<agentID>` is replaced with `environment` and it has no goal. So formally, an `environment_module>` is declared as follows:

```
module environment controls <controlled_vars>
init
:: true ~> <action>;
...
update
:: <guard> ~> <action>;
...
```

3. **Property declaration.** For E-NASH and A-NASH, we need to specify the property that needs to be checked. This property is expressed in LTL formula. The formal declaration is as follows

```
property
::  <LTL_formula>;
```

where `<LTL_formula>` syntax is as shown in Table 2.

An Example

This example is taken from [3]. Consider a peer-to-peer network with two agents. At each time step, each agent can only either tries to download or to upload. In order to download successfully, an agent must download while the other uploads at the same time. Both agent want to download infinitely often. The property to be checked is “each agent can download infinitely often”, expressed in LTL formula $(\mathsf{G}\,\mathsf{F}\,d_a) \wedge (\mathsf{G}\,\mathsf{F}\,d_b)$. This system can be expressed in SRML as follows; right side column is the translation into SRML code.

module m_a **controls** u_a, d_a
init
$:: \top \rightsquigarrow u_a' := \top, d_a' := \bot;$
$:: \top \rightsquigarrow u_a' := \bot, d_a' := \top;$
update
$:: \top \rightsquigarrow u_a' := \top, d_a' := \bot;$
$:: \top \rightsquigarrow u_a' := \bot, d_a' := \top;$
goal
$:: \mathsf{G}\,\mathsf{F}\,(d_a \wedge u_b);$

module m_b **controls** u_b, d_b
init
$:: \top \rightsquigarrow u_b' := \top, d_b' := \bot;$
$:: \top \rightsquigarrow u_b' := \bot, d_b' := \top;$
update
$:: \top \rightsquigarrow u_b' := \top, d_b' := \bot;$
$:: \top \rightsquigarrow u_b' := \bot, d_b' := \top;$
goal
$:: \mathsf{G}\,\mathsf{F}\,(d_b \wedge u_a);$

property
$:: (\mathsf{G}\,\mathsf{F}\,d_a) \wedge (\mathsf{G}\,\mathsf{F}\,d_b);$

```
module ma controls ua, da
  init
  ::  true ~> ua':= true, da':= false;
  ::  true ~> ua':= false, da':= true;
  update
  ::  true ~> ua':= true, da':= false;
  ::  true ~> ua':= false, da':= true;
  goal
  ::  G F (da and ub);

module mb controls ub, db
  init
  ::  true ~> ub':= true, db':= false;
  ::  true ~> ub':= false, db':= true;
  update
  ::  true ~> ub':= true, db':= false;
  ::  true ~> ub':= false, db':= true;
  goal
  ::  G F (db and ua);

property
::  (G F da) and (G F db);
```

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, September 2002.

[2] Julian Gutierrez, Paul Harrenstein, and Michael Wooldridge. From model checking to equilibrium checking: Reactive modules for rational verification. *Artificial Intelligence*, 248:123–157, 2017.

[3] Alexis Toumi, Julian Gutierrez, and Michael Wooldridge. A tool for the automated verification of nash equilibria in concurrent games. In *ICTAC*, volume 9399 of *LNCS*, pages 583–594, Cali, Colombia, 2015. Springer.