Model Translation from Papyrus-RT into the nuXmv Model Checker

Ms. Sneha Sahu

2020 September 15
Agenda

Introduction

• Overview
• Tools

Translation

• Approach
• Case Study Analysis

Conclusion
Introduction

- System Requirements
- UML-RT model (Papyrus-RT)
- Executable CDT Project
Introduction

- System Requirements
- UML-RT model (Papyrus-RT)
- Executable CDT Project
- M2M, M2T
- FSM
- nuXmv model
Papyrus-RT

- Basic Modelling Components – Capsules, Protocols, Ports, Attributes, Triggers, Guards, Composite Structure, State Diagram

- Port communication via controller message queue

- Run-To-Completion execution semantics
nuXmv Model Checker

- Module Structure –
  - VARiable declarations
  - INITializations (single assignment constraint)
  - ASSINGments and TRANSitions (circular dependency check)
- Only synchronous systems unlike NuSMV
- BDD and SAT based verification of Finite State Machines
- SMT-based techniques for verification of Infinite State Machines
## Translation Approach

<table>
<thead>
<tr>
<th>UML-RT</th>
<th>NU\text{\textbullet}MV</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Diagrams</td>
<td>FSMs</td>
</tr>
<tr>
<td>Non trivial transition effects</td>
<td>Abstract out ‘what’, modify ‘how’</td>
</tr>
<tr>
<td>Top Capsule</td>
<td>Main Module</td>
</tr>
<tr>
<td>Asynchronous Communication*</td>
<td>Controller logic in Main Module</td>
</tr>
<tr>
<td>Timing Protocol*</td>
<td>Countdown logic in Separate module</td>
</tr>
<tr>
<td>Other Capsules &amp; Protocols</td>
<td>Separate modules with input parameters</td>
</tr>
<tr>
<td>Ports</td>
<td>VAR declarations</td>
</tr>
<tr>
<td>Connections</td>
<td>as inputs to Modules</td>
</tr>
</tbody>
</table>

* These algorithm are defined based on the gained understanding of the Papyrus-RT system
SD → FSM, abstraktion

**Code Snippet**

```cpp
int random = rand() % 3;
play.picked(static_cast<Choice>(random)).send();
```

**ASSIGN**

```cpp
next(msg_Play) := case
t2b: picked;
    TRUE: null;
esac;
next(param_Play) := case
t2b: {1, 2, 3};
    TRUE: 0;
esac;
```
Capsules, Protocols → Modules

```plaintext
MODULE main
VAR
reservoir: capsule_reservoir(TURN_RES, port_button, bgc.port_insulinAsk);
...
```

```plaintext
MODULE capsule_reservoir(myTurn, in_res, in_insulinInject)
VAR
port_res: protocol_commands(TRUE, msg_res, 0);
msg_res: (null, fail);
port_insulinInject: protocol_insulin(TRUE, msg_insulinInject, param_insulinInject);
msg_insulinInject: (null, inject);
param_insulinInject: 0..3;
rlvl: const_rlvl_min .. const_rlvl_max;
state: {p_initial, s_start, s_ok, wait_t3_ok, wait_t4_ok, wait_t5_ok, s_stop};
inputDose: 0..3;
```

```plaintext
DEFINE
WAITING := (state = wait_t3_ok | state = wait_t4_ok | state = wait_t5_ok);
const_rlvl_max := 100;
const_rlvl_min := 0;
```
Capsules, Protocols → Modules

```
MODULE main
VAR ...
  reservoir: capsule_reservoir(TURN_RES, port_button, bgc.port_insulinAsk);
...
```

```
MODULE capsule_reservoir(myTurn, in_res, in_insulinInject)
VAR
  port_res: protocol_commands(TRUE, msg_res, 0);
  msg_res: (null, fail);
  port_insulinInject: protocol_insulin(TRUE, msg_insulinInject, param_insulinInject);
  msg_insulinInject: (null, inject);
  param_insulinInject: 0..3;
  rLvl: const_rLvl_min .. const_rLvl_max;
  state: {p_initial, s_start, s_ok, wait_t3_ok, wait_t4_ok,
```

```
MODULE protocol_insulin(conjugated, msg, param)
DEFINE
  out_ask := (conjugated = FALSE & msg = ask);
  out_ask_doseComputed := (out_ask ? param : 0);
  in_inject := (conjugated = TRUE & msg = inject);
  in_inject_doseGiven := (in_inject ? param : 0);
```

Wissen durch Praxis stärkt
Additional rules to handle ‘empty FSM’ situation

[Consider an FSM with 3 transitions – t1, t2, t3]

❑ (less than zero case)
   An extra transition ‘t_none’ is defined, to take care of any undefined transition.
   
   ```
   DEFINE t_none := !(t1 | t2 | t3); TRANS t_none -> next(*) = *;
   ```

❑ (more than 1 case)
   Avoid non-deterministic FSM, by ensuring mutual exclusion of all outgoing transitions from any particular state

   ```
   t2 := (state = s1 & trigger_a & guard_b);
   t3 := (state = s1 & trigger_c);
   -- issue: non determinism if both trigger_a and trigger_c are true simultaneously
   t2 := (state = s1 & !trigger_c & trigger_a & guard_b);
   t3 := (state = s1 & trigger_c);
   ```

❑ Add an INVARIANT to check that exactly one transition is valid at any instance.

   ```
   INVAR ( t_none | (t1 & !t2 & !t3) | (!t1 & t2 & !t3) | (!t1 & !t2 & t3) )
   ```
Ping Pong Model

- Single controller queue, with 2 units, communicating directly
- Only one message in queue at any instance of time
- Complexities: none

Properties Verification –
- “A ‘ping’ must be followed by a ‘pong’ and a ‘pong’ must be followed by a ‘ping’”
  \[ \text{SPEC AG (MSG\_PING} \rightarrow \text{ A [ AX !MSG\_PING U MSG\_PONG] & AF MSG\_PONG)} \]
- “communication starts with a ‘ping’”
  \[ \text{SPEC A [ (!MSG\_PING & !MSG\_PONG) U MSG\_PING]} \]
Ping Pong Model

Papyrus-RT

nuXmv

Wissen durch Praxis stärkt

Slide 13
Rock Paper Scissor Model

- Single controller queue, with 3 units
- Possibility of more than one message in queue at any instance of time
- Complexities:
  - Multiplicity of capsules and ports
  - Connections carry payload
  - Hierarchical State Diagram
  - Non-trivial code snippets
Rock Paper Scissor Model

- Properties Verification – each Player:
  - “a `player’ always eventually waits in ‘idle’ mode for a signal from the `referee' to be able to play its round”
    \[\text{SPEC AG AF ( A [state=s_idle U inPlay.in_go] )}\]
  - “Whenever a player receives a 'go', it eventually sends out a choice from the given domain and goes back to 'idle’”
    \[\text{SPEC AG (inPlay.in_go } \rightarrow \text{ AF (param_Play in \{1,2,3\} } \rightarrow \text{ msg_Play=picked & state=s_idle) )}\]

```plaintext
-> State: 1.8 <-
  player1.state = wait_t2_idle
  player2.state = wait_t2_idle
  referee.msg_Play = null
  referee.param_Play = 0

-> State: 1.9 <-
  msgPtr_default = 1

-> State: 1.10 <-
  msgPtr_default = 0
  player1.msg_Play = picked
  player1.param_Play = 1
  player1.state = s_idle

-> State: 1.11 <-
  msgPtr_default = 2
...

-> State: 1.12 <-
  player2.msg_Play = picked
  player2.param_Play = 2
  player2.state = s_idle
```
Insulin Pump Model

- **Multiple** controller queue, with 4 units
- Possibility of more than one message in queue at any instance of time
- **Complexity:**
  - Multiplicity of ports
  - Connections with/without payload
  - Multi-threading
  - Non-trivial code snippets
  - Multiple overlapping timer reset requests
Insulin Pump Model

- Properties Verification – BG controller unit:
  - “The injected insulin dose should always be within the prescribed maximum limit”
    \[ \text{SPEC AG} \ (\text{inputDose} \leq \text{const\_maxAllowedDose}) \]
  - “injected dose is same as the requested amount”
    \[ \text{LTLSPEC G} \ \text{in\_insulinAsk.in\_inject} \rightarrow \text{prop\_drugAsked\_equals\_drugToGive} \]
  - “when the blood glucose is in the normal or lower range, computed dose must always be zero”
    \[ \text{SPEC AG} \ ((\text{prop\_bg\_low} \mid \text{prop\_bg\_normal}) \rightarrow \text{computedDose} = 0) \]
Insulin Pump Model

- Properties Verification – system:
  - “a `Running' system should eventually succeed in keeping blood glucose within the normal range”
    
    \[
    \text{CTLSPEC AG} \ (\text{prop\_system\_running} \rightarrow \text{AF} \ (\text{bgc\_prop\_bg\_normal} \ |
    \text{prop\_system\_stopped}))
    \]
  - “system stops only when either battery or reservoir fail”
    
    \[
    \text{SPEC AG} \ \text{prop\_system\_running} \rightarrow \text{A} \ [ \text{AX} \ \text{prop\_system\_running} \ \text{U} \ (!\text{battery\_prop\_ok} \ |
    !\text{reservoir\_prop\_ok})]
    \]
  - “when system is stopped, ensure all modules stop eventually and remain so”
    
    \[
    \text{SPEC AG} \ (\text{prop\_system\_stopped} \rightarrow \text{AF AG} \ (\text{battery\_prop\_stopped} \ & \ \text{reservoir\_prop\_stopped} \ &
    \text{bgc\_prop\_stopped}))
    \]
Insulin Pump Model

Wissen durch Praxis stärkt
Insulin Pump Model

```python
# Blood Glucose: previous = 130, current = 130
# battery status: 17
# battery status: 16
# 'bgc' sending 'insulin.ask(1)' --> 'reservoir'.
# 'reservoir' received 'insulin.ask(1)'.
# 'reservoir' sending 'inject(1)' --> 'bgc'.
# 'reservoir' sending 'fail' --> ... !!
# 'bgc' received 'insulin.inject(1)'.
# Blood Glucose: previous = 130, current = 125

> Stopping Pump
> ... sending 'stop' --> 'battery'.
> ... sending 'stop' --> 'reservoir'.
> ... sending 'stop' --> 'bgc'.
# insulin level: 19
# 'reservoir' received 'stop'.
- Reservoir Stopped [19 units]
# 'battery' received 'stop'.
- Battery Stopped [16 units]
# 'bgc' received 'stop'.
- BGC Stopped [BG = 125 units]
```

```python
-> State: 2.118 <-
  ...
  battery.blvl = 11
  battery.state = s_stop
  ...
  reservoir.rlvl = 19
  reservoir.state = s_stop
  ...
  bgc.state = s_stop
  bgc.bglvl = 125
  bgc.bglvl_prev = 130
  ...
  STATE = s_stopping
```
Conclusion and Future Works

- Detailed Translation Mapping
- Mechanical application on multiple case studies
- Works in coordination with certain modelling restrictions –
  - Message payload only as integer type and restricted to a max count of 1
  - Incoming payload to be stored locally before use
  - An attribute is modified only once between 2 stable states
- Timer related issues for Multithreaded system designs, to be improvised
- Handling of enumerations and more as payload
- Reverse Translation from nuXmv counter traces, for model correction
References


Thank you for your time!!