Towards Model Checking Executable UML Specifications in mCRL2

Helle Hvid Hansen Jeroen Ketema Bas Luttik
MohammadReza Mousavi Jaco van de Pol

Eindhoven University of Technology
University of Twente

8 December 2009

Introduction

Verification of railway safety systems (interlockings):

- Specification is highly declarative (not an implementation)
- Specification written in *executable UML*

Executable UML (xUML):

- Class diagrams and state machines
- Particular dialect comes with a simulator

Verification approach:

- Instantiate the model based on the layout of a railway yard
- Transform into an mCRL2 specification (process algebra)
- Apply model checking (both explicit and symbolic)

Introduction

Verification of railway safety systems (interlockings):

- Specification is highly declarative (not an implementation)
- Specification written in *executable UML*

Executable UML (xUML):

- Class diagrams and state machines
- Particular dialect comes with a simulator

Verification approach:

- Instantiate the model based on the layout of a railway yard
- Transform into an mCRL2 specification (process algebra)
- Apply model checking (both explicit and symbolic)

Introduction

Verification of railway safety systems (interlockings):

- Specification is highly declarative (not an implementation)
- Specification written in *executable UML*

Executable UML (xUML):

- Class diagrams and state machines
- Particular dialect comes with a simulator

Verification approach:

- Instantiate the model based on the layout of a railway yard
- Transform into an mCRL2 specification (process algebra)
- Apply model checking (both explicit and symbolic)

mCRL2

mCRL2 is an ACP-based process algebra:

- Synchronous communication between processes
- Processes and actions may carry data

Data types

- Built-in data types: integers, lists, ...
- Abstract data types: $\texttt{sort myState} = \texttt{struct Yes} \mid \texttt{No}$

Sequential processes (with data)

- Recursive processes: $\texttt{proc A}(\ldots) = \cdots \texttt{A}(\ldots) \cdots ;$
- Actions: $\texttt{a}(\ldots)$
- Sequential and alternative composition: $.$ and $+$
- If-then-else construct: $c \rightarrow s \diamond t$

mCRL2

mCRL2 is an ACP-based process algebra:

- Synchronous communication between processes
- Processes and actions may carry data

Data types

- Built-in data types: integers, lists, ...
- Abstract data types: `sort myState = struct Yes | No`

Sequential processes (with data)

- Recursive processes: $\texttt{proc}\ \texttt{A}(\ldots) = \cdots \texttt{A}(\ldots) \cdots$;
- Actions: a(...)
- Sequential and alternative composition: . and +
- If-then-else construct: $c \rightarrow s \diamond t$

mCRL2

mCRL2 is an ACP-based process algebra:

- Synchronous communication between processes
- Processes and actions may carry data

Data types

- Built-in data types: integers, lists, ...
- Abstract data types: `sort myState = struct Yes | No`

Sequential processes (with data)

- Recursive processes: `proc A(...) =` $\cdots$ `A(...)` $\cdots$;
- Actions: a(...)
- Sequential and alternative composition: . and $+$
- If-then-else construct: $c \rightarrow s \diamond t$

mCRL2 (cont.)

Sequential processes (cont.)

- Quantification over data: $\sum_{d:D} P(d)$

Example

$\mathtt{proc}\ \mathtt{A}(n:\mathbb{N}) = \sum_{m:\mathbb{N}} \mathtt{a}(m).(m = 0) \rightarrow \mathtt{A}(n+1) \diamond (\mathtt{A}(m) + \mathtt{A}(n))$

Parallel processes and communication

- parallel composition: $\|$
- synchronous communication (multi-actions): $\mathtt{a}_1 | \cdots | \mathtt{a}_n \rightarrow \mathtt{b}$

Example

$\mathtt{comm}(\{\mathtt{a}|\mathtt{b} \rightarrow \mathtt{c}\}, \mathtt{a}(m) \,\|\, \mathtt{b}(m))$ we observe $\mathtt{c}(m)$

mCRL2 (cont.)

Sequential processes (cont.)

- Quantification over data: $\sum_{d:D} P(d)$

Example

$\mathtt{proc}\ \mathtt{A}(n:\mathbb{N}) = \sum_{m:\mathbb{N}} \mathtt{a}(m).(m = 0) \rightarrow \mathtt{A}(n+1) \diamond (\mathtt{A}(m) + \mathtt{A}(n))$

Parallel processes and communication

- parallel composition: $\|$
- synchronous communication (multi-actions): $\mathtt{a}_1 | \cdots | \mathtt{a}_n \rightarrow \mathtt{b}$

Example

$\mathtt{comm}(\{\mathtt{a}|\mathtt{b} \rightarrow \mathtt{c}\}, \mathtt{a}(m) \,\|\, \mathtt{b}(m))$ we observe $\mathtt{c}(m)$

mCRL2 (cont.)

Sequential processes (cont.)

- Quantification over data: $\sum_{d:D} P(d)$

Example

$\mathtt{proc}\ \mathtt{A}(n:\mathbb{N}) = \sum_{m:\mathbb{N}} \mathtt{a}(m).(m = 0) \rightarrow \mathtt{A}(n+1) \diamond (\mathtt{A}(m) + \mathtt{A}(n))$

Parallel processes and communication

- parallel composition: $\|$
- synchronous communication (multi-actions): $\mathtt{a}_1 | \cdots | \mathtt{a}_n \rightarrow \mathtt{b}$

Example

$\mathtt{comm}(\{\mathtt{a}|\mathtt{b} \rightarrow \mathtt{c}\}, \mathtt{a}(m) \,\|\, \mathtt{b}(m))$ we observe $\mathtt{c}(m)$

mCRL2 (cont.)

Sequential processes (cont.)

- Quantification over data: $\sum_{d:D} P(d)$

Example

$\mathtt{proc}\ \mathtt{A}(n:\mathbb{N}) = \sum_{m:\mathbb{N}} \mathtt{a}(m).(m = 0) \rightarrow \mathtt{A}(n+1) \diamond (\mathtt{A}(m) + \mathtt{A}(n))$

Parallel processes and communication

- parallel composition: $\|$
- synchronous communication (multi-actions): $\mathtt{a}_1 | \cdots | \mathtt{a}_n \rightarrow \mathtt{b}$

Example

$\mathtt{comm}(\{\mathtt{a}|\mathtt{b} \rightarrow \mathtt{c}\}, \mathtt{a}(m) \parallel \mathtt{b}(m))$ we observe $\mathtt{c}(m)$

xUML Constructs

Only translate constructs that occur in railway specifications

- Class diagrams
- Inheritance and associations between classes
- No association classes (classes labelling associations)
- (Nested) state machines

xUML Constructs

Only translate constructs that occur in railway specifications

Class diagrams

Inheritance and associations between classes

No association classes (classes labelling associations)

(Nested) state machines

- States:
  - Concurrent and composite states (AND- and OR-states)
  - Initial pseudo states (no history and final pseudo states)
- Transitions labelled with "trigger[condition]/action"-triples
  - *Trigger* needed to take the transition (signal or change event)
  - *Condition* needed to be valid upon taking the transition
  - *Action* to be executed upon taking the transition

xUML Constructs

Only translate constructs that occur in railway specifications

Class diagrams

Inheritance and associations between classes

No association classes (classes labelling associations)

(Nested) state machines

- States:
  - Concurrent and composite states (AND- and OR-states)
  - Initial pseudo states (no history and final pseudo states)
- Transitions labelled with "trigger[condition]/action"-triples
  - *Trigger* needed to take the transition (signal or change event)
  - *Condition* needed to be valid upon taking the transition
  - *Action* to be executed upon taking the transition

xUML Constructs

Only translate constructs that occur in railway specifications

Class diagrams

Inheritance and associations between classes

No association classes (classes labelling associations)

(Nested) state machines

- States:
  - Concurrent and composite states (AND- and OR-states)
  - Initial pseudo states (no history and final pseudo states)
- Transitions labelled with "trigger[condition]/action"-triples
  - *Trigger* needed to take the transition (signal or change event)
  - *Condition* needed to be valid upon taking the transition
  - *Action* to be executed upon taking the transition

Signal and Change Events

Events are stored in event pools (buffers), one per class instance

Signal events

Signals can be sent to classes and their associated state machines

- Signals are sent asynchronously
- Once received signal event is added to an event pool

Change events

Change events are of the form

$$\texttt{when}(\mathit{cond})$$

where *cond* is a boolean expression:

- The event is added to an event pool when *cond* becomes valid
- The event is not removed once *cond* becomes invalid again

Signal and Change Events

Events are stored in event pools (buffers), one per class instance

Signal events

Signals can be sent to classes and their associated state machines

- Signals are sent asynchronously
- Once received signal event is added to an event pool

Change events

Change events are of the form

$$\texttt{when}(\mathit{cond})$$

where *cond* is a boolean expression:

- The event is added to an event pool when *cond* becomes valid
- The event is not removed until *cond* becomes invalid again

Signal and Change Events

Events are stored in event pools (buffers), one per class instance

Signal events

Signals can be sent to classes and their associated state machines

- Signals are sent asynchronously
- Once received signal event is added to an event pool

Change events

Change events are of the form

$$\texttt{when}(\mathit{cond})$$

where *cond* is a boolean expression:

- The event is added to an event pool when *cond* becomes valid
- The event is not removed once *cond* becomes invalid again

Signal and Change Events

Events are stored in event pools (buffers), one per class instance

Signal events

Signals can be sent to classes and their associated state machines

- Signals are sent asynchronously
- Once received signal event is added to an event pool

Change events

Change events are of the form

$$\texttt{when}(\mathit{cond})$$

where *cond* is a boolean expression:

- The event is added to an event pool when *cond* becomes valid
- The event is not removed once *cond* becomes invalid again

Signal and Change Events

Events are stored in event pools (buffers), one per class instance

Signal events

Signals can be sent to classes and their associated state machines

- Signals are sent asynchronously
- Once received signal event is added to an event pool

Change events

Change events are of the form

$$\texttt{when}(\mathit{cond})$$

where *cond* is a boolean expression:

- The event is added to an event pool when *cond* becomes valid
- The event is not removed once *cond* becomes invalid again

Run-to-Completion Assumptions

Run-to-completion assumptions specify the allowed interleavings

Definition (Run-to-completion (RTC))

Local RTC All actions of a transition in a *state machine* S are executed before a new transition is taken by S

Atomic RTC All actions of a transition in the *system* are executed before any new transition is taken

Global RTC External events are only accepted by the system in case (i) all event pools are empty and (ii) no actions are being executed

Local RTC is minimally required by the UML standard

The available simulator enforces both atomic and global RTC

Run-to-Completion Assumptions

Run-to-completion assumptions specify the allowed interleavings

Definition (Run-to-completion (RTC))

Local RTC All actions of a transition in a *state machine* S are executed before a new transition is taken by S

Atomic RTC All actions of a transition in the *system* are executed before any new transition is taken

Global RTC External events are only accepted by the system in case (i) all event pools are empty and (ii) no actions are being executed

Local RTC is minimally required by the UML standard

The available simulator enforces both atomic and global RTC

Run-to-Completion Assumptions

Run-to-completion assumptions specify the allowed interleavings

Definition (Run-to-completion (RTC))

Local RTC All actions of a transition in a *state machine S* are executed before a new transition is taken by *S*

Atomic RTC All actions of a transition in the *system* are executed before any new transition is taken

Global RTC External events are only accepted by the system in case (i) all event pools are empty and (ii) no actions are being executed

Local RTC is minimally required by the UML standard

The available simulator enforces both atomic and global RTC

Run-to-Completion Assumptions

Run-to-completion assumptions specify the allowed interleavings

Definition (Run-to-completion (RTC))

Local RTC All actions of a transition in a *state machine S* are executed before a new transition is taken by *S*

Atomic RTC All actions of a transition in the *system* are executed before any new transition is taken

Global RTC External events are only accepted by the system in case (i) all event pools are empty and (ii) no actions are being executed

Local RTC is minimally required by the UML standard

The available simulator enforces both atomic and global RTC

Run-to-Completion Assumptions

Run-to-completion assumptions specify the allowed interleavings

Definition (Run-to-completion (RTC))

Local RTC All actions of a transition in a *state machine S* are executed before a new transition is taken by *S*

Atomic RTC All actions of a transition in the *system* are executed before any new transition is taken

Global RTC External events are only accepted by the system in case (i) all event pools are empty and (ii) no actions are being executed

Local RTC is minimally required by the UML standard

The available simulator enforces both atomic and global RTC

Translating Class Diagrams

Each class is represented by a process:

- Inheritance is dealt with by "flattening" the class hierarchy
  - ⇒ Concurrent composition of state machines related to classes
- Associations become parameters of processes

Example

Translating Class Diagrams

Each class is represented by a process:

- Inheritance is dealt with by "flattening" the class hierarchy
  - $\Rightarrow$ Concurrent composition of state machines related to classes
- Associations become parameters of processes

Example

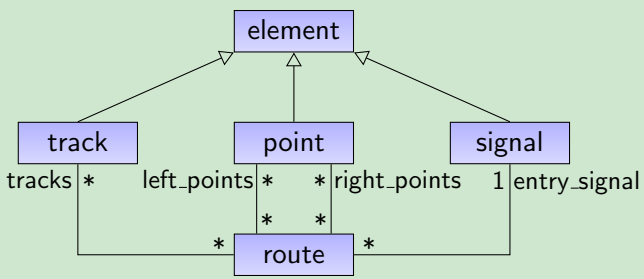

```
proc track(routes : ...) = ...;

proc point(routes : ...) = ...;

proc signal(routes : ...) = ...;

proc route(tracks : ..., left_points : ...
           right_points : ..., entry_signal : ...) = ...;
```

Translating Class Diagrams

Each class is represented by a process:

- Inheritance is dealt with by "flattening" the class hierarchy
  - $\Rightarrow$ Concurrent composition of state machines related to classes
- Associations become parameters of processes

Example



$$\mathtt{proc\ track(routes : \ldots) = \ldots;}$$

$$\mathtt{proc\ point(routes : \ldots) = \ldots;}$$

$$\mathtt{proc\ signal(routes : \ldots) = \ldots;}$$

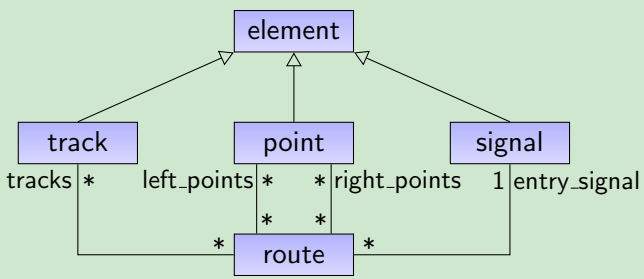
$$\mathtt{proc\ route(tracks : \ldots,\ left\_points : \ldots,}$$
$$\mathtt{right\_points : \ldots,\ entry\_signal : \ldots) = \ldots;}$$

Translating Class Diagrams

Each class is represented by a process:

- Inheritance is dealt with by "flattening" the class hierarchy
  - $\Rightarrow$ Concurrent composition of state machines related to classes
- Associations become parameters of processes

Example



$$\texttt{proc track}(\texttt{routes} : \ldots) = \ldots;$$

$$\texttt{proc point}(\texttt{routes} : \ldots) = \ldots;$$

$$\texttt{proc signal}(\texttt{routes} : \ldots) = \ldots;$$

$$\texttt{proc route}(\texttt{tracks} : \ldots, \texttt{left\_points} : \ldots, \texttt{right\_points} : \ldots, \texttt{entry\_signal} : \ldots) = \ldots;$$

Translating Event Pools and State Machines

Each process representing a class consists of two parallel processes:

- Buffer process
  - ⇒ Represents event pool associated with an instance of a class
  - ⇒ Asynchronous communication in synchronous environment
- Process representing the state machine related to the class
  - ⇒ States are represented as data parameters to the process
  - ⇒ Process is a message loop:
    get message, execute actions, update state, get message, ...

Example

Translating Event Pools and State Machines

Each process representing a class consists of two parallel processes:

- Buffer process
  - ⇒ Represents event pool associated with an instance of a class
  - ⇒ Asynchronous communication in synchronous environment
- Process representing the state machine related to the class
  - ⇒ States are represented as data parameters to the process
  - ⇒ Process is a message loop:
    get message, execute actions, update state, get message, . . .

Example

$$\texttt{proc M}(S:\ldots,\ \texttt{Auto\_S}:\ldots) =$$
$$\sum_{m:\texttt{Msg}} \texttt{get\_msg}(m).$$
$$(S = \texttt{manual}\ \&\&\ m = \texttt{automatic}) \rightarrow$$
$$\texttt{M(automatic, not\_ready)}$$
$$\diamond \cdots ;$$

Translating Event Pools and State Machines

Each process representing a class consists of two parallel processes:

- Buffer process
  - ⇒ Represents event pool associated with an instance of a class
  - ⇒ Asynchronous communication in synchronous environment
- Process representing the state machine related to the class
  - ⇒ States are represented as data parameters to the process
  - ⇒ Process is a message loop:
    get message, execute actions, update state, get message, ...

Example

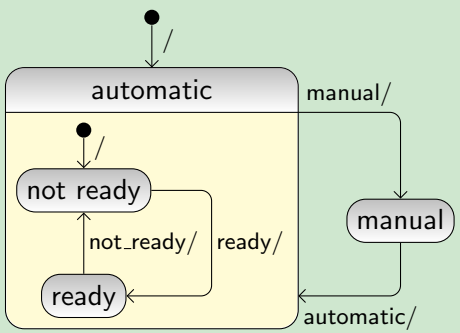
$$\mathtt{proc}\ \mathtt{M}(\mathtt{S}:\ldots,\ \mathtt{Auto\_S}:\ldots) =$$
$$\sum_{m:\mathtt{Msg}} \mathtt{get\_msg}(m).$$
$$(\mathtt{S} == \mathtt{manual}\ \&\&\ m == \mathtt{automatic}) \Rightarrow$$
$$\mathtt{M}(\mathtt{automatic},\ \mathtt{not\_ready})$$
$$\diamond \ldots ;$$

Translating Event Pools and State Machines

Each process representing a class consists of two parallel processes:

- Buffer process
  - ⇒ Represents event pool associated with an instance of a class
  - ⇒ Asynchronous communication in synchronous environment
- Process representing the state machine related to the class
  - ⇒ States are represented as data parameters to the process
  - ⇒ Process is a message loop:
    get message, execute actions, update state, get message, ...

Example



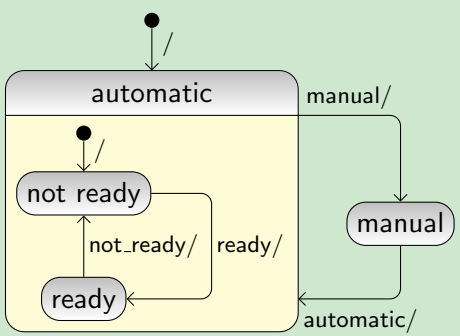
$$\begin{aligned}&\mathtt{proc}\ \mathtt{M}(\mathtt{S}:\ldots,\ \mathtt{Auto\_S}:\ldots) =\\ &\quad \textstyle\sum_{m:\mathtt{Msg}} \mathtt{get\_msg}(m).\\ &\qquad (\mathtt{S} = \mathtt{manual}\ \&\&\ m = \mathtt{automatic}) \rightarrow\\ &\qquad\quad \mathtt{M}(\mathtt{automatic},\ \mathtt{not\_ready})\\ &\quad \diamond\ \ldots\ ;\end{aligned}$$

Translating Event Pools and State Machines

Each process representing a class consists of two parallel processes:

- Buffer process
  - ⇒ Represents event pool associated with an instance of a class
  - ⇒ Asynchronous communication in synchronous environment
- Process representing the state machine related to the class
  - ⇒ States are represented as data parameters to the process
  - ⇒ Process is a message loop:
    get message, execute actions, update state, get message, ...

Example



$$\begin{aligned}&\texttt{proc}\ \texttt{M}(\texttt{S}:\ldots,\ \texttt{Auto\_S}:\ldots) = \\ &\quad \textstyle\sum_{m:\texttt{Msg}} \texttt{get\_msg}(m). \\ &\quad\ (\texttt{S} = \texttt{manual}\ \&\&\ m = \texttt{automatic}) \rightarrow \\ &\quad\quad \texttt{M}(\texttt{automatic},\ \texttt{not\_ready}) \\ &\quad \diamond\ \ldots\ ;\end{aligned}$$

Translating Change Events

Change events are translated by introducing "monitor" processes

- One monitor per occurring change event
- Inner workings:
  1. If part of the state referred to by the change event changes, then a message is sent synchronously to the related monitor
  2. Monitor checks if condition is valid while it wasn't before
  3. If so, the state machine to which the event belongs is notified (message is put in buffer associated with the state machine)

Observations (without showing any mCRL2 specification):

- Monitors duplicate state data
- Communication with monitors increases number of transitions

Translating Change Events

Change events are translated by introducing "monitor" processes

- One monitor per occurring change event
- Inner workings:
  1. If part of the state referred to by the change event changes, then a message is sent synchronously to the related monitor
  2. Monitor checks if condition is valid while it wasn't before
  3. If so, the state machine to which the event belongs is notified (message is put in buffer associated with the state machine)

Observations (without showing any mCRL2 specification):

- Monitors duplicate state data
- Communication with monitors increases number of transitions

Translating Change Events

Change events are translated by introducing "monitor" processes

- One monitor per occurring change event
- Inner workings:
  1. If part of the state referred to by the change event changes, then a message is sent synchronously to the related monitor
  2. Monitor checks if condition is valid while it wasn't before
  3. If so, the state machine to which the event belongs is notified (message is put in buffer associated with the state machine)

Observations (without showing any mCRL2 specification):

- Monitors duplicate state data
- Communication with monitors increases number of transitions

Translating Change Events

Change events are translated by introducing "monitor" processes

- One monitor per occurring change event
- Inner workings:
  1. If part of the state referred to by the change event changes, then a message is sent synchronously to the related monitor
  2. Monitor checks if condition is valid while it wasn't before
  3. If so, the state machine to which the event belongs is notified (message is put in buffer associated with the state machine)

Observations (without showing any mCRL2 specification):

- Monitors duplicate state data
- Communication with monitors increases number of transitions

Model Checking

Remark

- Unlimited buffer size in translation ⇒ Infinite state space
- Only local RTC in translation ⇒ Starvation

Mitigation:

- Limited buffer space (solves *only* infinite state space problem)
- Barrier synchronisation (solves both issues, but global RTC)

Small Toy Specification (7 class instances)

Version	State space	Symbolic	Explicit
buffer size 1	61×10^{12}	113 secs	not feasible
barrier sync	8×10^{6}	160 secs	$9\frac{1}{2}$ minutes

No atomic RTC: yields traces not observable in the simulator

Model Checking

Remark

- Unlimited buffer size in translation ⇒ Infinite state space
- Only local RTC in translation ⇒ Starvation

Mitigation:

- Limited buffer space (solves *only* infinite state space problem)
- Barrier synchronisation (solves both issues, but global RTC)

Small Toy Specification (7 class instances)

Version	State space	Symbolic	Explicit
buffer size 1	61×10^{12}	113 secs	not feasible
barrier sync	8×10^{6}	160 secs	$9\frac{1}{2}$ minutes

No atomic RTC: yields traces not observable in the simulator

Model Checking

Remark

- Unlimited buffer size in translation ⇒ Infinite state space
- Only local RTC in translation ⇒ Starvation

Mitigation:

- Limited buffer space (solves *only* infinite state space problem)
- Barrier synchronisation (solves both issues, but global RTC)

Small Toy Specification (7 class instances)

Version	State space	Symbolic	Explicit
buffer size 1	61×10^{12}	113 secs	not feasible
barrier sync	8×10^{6}	160 secs	$9\frac{1}{2}$ minutes

No atomic RTC: yields traces not observable in the simulator

Model Checking

Remark

- Unlimited buffer size in translation $\Rightarrow$ Infinite state space
- Only local RTC in translation $\Rightarrow$ Starvation

Mitigation:

- Limited buffer space (solves *only* infinite state space problem)
- Barrier synchronisation (solves both issues, but global RTC)

Small Toy Specification (7 class instances)

Version	State space	Symbolic	Explicit
buffer size 1	61×10^{12}	113 secs	not feasible
barrier sync	8×10^{6}	160 secs	$9\frac{1}{2}$ minutes

No atomic RTC: yields traces not observable in the simulator

Model Checking

Remark

- Unlimited buffer size in translation $\Rightarrow$ Infinite state space
- Only local RTC in translation $\Rightarrow$ Starvation

Mitigation:

- Limited buffer space (solves *only* infinite state space problem)
- Barrier synchronisation (solves both issues, but global RTC)

Small Toy Specification (7 class instances)

Version	State space	Symbolic	Explicit
buffer size 1	61×10^{12}	113 secs	not feasible
barrier sync	8×10^{6}	160 secs	$9\frac{1}{2}$ minutes

No atomic RTC: yields traces not observable in the simulator

Conclusion

Translation from xUML to mCRL2:

- Not extremely difficult
- Except for change events (not completely satisfactory)

Model checking the translation:

- Measures needed to avoid infinite state space and starvation
- State space can be huge
- Traces depend on RTC assumptions (different for simulator)

Future work:

- Automatic translation using the Epsilon framework
- Extend the translation to other xUML constructs
- Re-consider the translation of change events (avoid them?)

Conclusion

Translation from xUML to mCRL2:

- Not extremely difficult
- Except for change events (not completely satisfactory)

Model checking the translation:

- Measures needed to avoid infinite state space and starvation
- State space can be huge
- Traces depend on RTC assumptions (different for simulator)

Future work:

- Automatic translation using the Epsilon framework
- Extend the translation to other xUML constructs
- Re-consider the translation of change events (avoid them?)

Conclusion

Translation from xUML to mCRL2:

- Not extremely difficult
- Except for change events (not completely satisfactory)

Model checking the translation:

- Measures needed to avoid infinite state space and starvation
- State space can be huge
- Traces depend on RTC assumptions (different for simulator)

Future work:

- Automatic translation using the Epsilon framework
- Extend the translation to other xUML constructs
- Re-consider the translation of change events (avoid them?)