The mCRL2 Toolset

Aad Mathijssen    Bas Ploeger

Design and Analysis of Systems Group / LaQuSo
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven

NXP Semiconductors

July 9th, 2008
Analysis techniques

Analysis techniques used in hardware/software development:

- **Structural analysis:** what *things* are in the system
  - Class diagrams
  - Component diagrams
  - Package diagrams
  - ...

- **Behavioural analysis:** what *happens* in the system
  - State diagrams
  - Message sequence charts
  - Petri nets
  - Process algebra
  - Temporal logic
  - ...

Behavioural analysis

What is it?

What is **behavioural analysis** about?

- **Modelling:**
  - Create an abstract model of the behaviour

- **Validation and Verification:**
  - Validation: does the model roughly behave as expected?
  - Verification: does the model satisfy the requirements in all states?
Behavioural analysis
Modelling

Why modelling?
 Behavioural analysis
 Modelling

 Why modelling?

 To reduce complexity:

 - Direct verification of all states of the software is impossible due to the huge number of states.
 - Much more complex than e.g. Rubik’s cube:

\[ 43,252,003,274,489,856,000 \times 4.3 \times 10^{19} \text{ states} \]
Behavioural analysis
Software lifecycle

Behavioural analysis is applicable to all phases of the software lifecycle:

- Requirements Analysis and Design:
  Prove that the design satisfies the requirements before anything is built.

- Implementation to Maintenance:
  Prove that the software satisfies the requirements in a rigorous way.
Behavioural analysis

Experience

From our experience:

- Without proper modelling it is **impossible** to get a system right.
- Implementing a model does **not** introduce substantial flaws.
- Modelling an implementation **nearly always** reveals flaws.
Behavioural analysis

Tool support

For verification of *industrial* systems, *tool support* is essential.

Toolsets for modelling, validation and verification of behaviour:

- CADP (INRIA Rhone Alpes, France)
- SPIN (Bell Labs, USA)
- FDR (Formal Systems Limited, Oxford, UK)
- Uppaal (Uppsala University, Sweden)
- NuSMV (Carnegie Mellon University, USA)
- mCRL2 (OAS group / LaQuSo, TU/e)
- ...
Toolset overview

History

History of the mCRL2 toolset:

- Late 1980s: Common Representation Language (CRL)
- From 1990: micro Common Representation Language ($\mu$CRL)
- During 1990s: development of the $\mu$CRL toolset
- Since 2004: micro Common Representation Language 2 (mCRL2) + toolset
Toolset overview
General information

- The mCRL2 toolset can be used for modelling, validation and verification of concurrent systems and protocols.
- Collection of tools
- Available for the following platforms:
  - Microsoft Windows
  - Linux
  - Mac OS X
  - FreeBSD
  - Solaris
- Distributed under the Boost license
- Available at http://mcrl2.org
Toolset overview

Tool categories

- Process specification
- Modal formula
- Lineariser
- PBES generator
- Parametrised boolean equation system
- Solvers
- Theorem proving
- Simulators
- LTS generator
- BES generator
- Labeled transition system
- Boolean equation system
- Visualisers

User input: abstract → concrete
Toolset overview
Linear process specifications

LPS tools:

- **Generation:**
  - `mcrll2lps`: Linearise a process specification

- **Information:**
  - `lpsinfo`: Information about an LPS
  - `lpspp`: Pretty prints an LPS

- **Simulation:**
  - `sim`: Text based simulation of an LPS
  - `xsim`: Graphical simulation of an LPS
Toolset overview
Linear process specifications (2)

LPS tools:
  - **Optimisation:**
    - `lpsconstelm`: Removes constant process parameters
    - `lpsparelm`: Removes irrelevant process parameters
    - `lpssuminst`: Instantiate sum operators
    - `lpssumelm`: Removes superfluous sum operators
    - `lpsactionrename`: Renaming of actions
    - `lpsconfcheck`: Marks confluent tau summands
    - `lpsinvelm`: Removes violating summands on invariants
    - `lpsbinary`: Replaces finite sort variables by vectors of boolean variables
    - `lpsrewr`: Rewrites data expressions of an LPS
    - `lpsuntime`: Removes time from an LPS
Toolset overview
Linear process specifications (3)

Simulation using \texttt{xsim}:
Toolset overview
Linear process specifications (3)

Simulation using \texttt{xsim} with plugins:
Toolset overview
Labelled transition systems

LTS tools:

- **Generation:**
  - `lps2lts`: Generates an LTS from an LPS

- **Information and visualisation:**
  - `ltsinfo`: Information about an LTS
  - `tracepp`: View traces generated by `sim/xsim` or `lps2lts`
  - `ltsgraph`: 2D LTS graph based visualisation
  - `ltsview`: 3D LTS state based clustered visualisation
  - `diagraphica`: Multivariate state visualisation and simulation analysis for LTSs

- **Comparison, conversion and minimisation:**
  - `ltscompare`: Compares two LTSs with respect to an equivalence or preorder
  - `ltsconvert`: Converts and minimises an LTS
Toolset overview
Labelled transition systems (2)

Visualisation using Itsgraph:
Toolset overview
Labelled transition systems (3)

Visualisation using Itsview:
Toolset overview
Labelled transition systems (4)

Visualisation using diagraphica:
Toolset overview
Parameterised boolean equation systems

PBES tools:
- **Generation:**
  - *lps2pbes*: Generates a PBES from an LPS and a temporal formula
  - *txt2pbes*: Parses a textual description of a PBES
- **Information:**
  - *pbesinfo*: Information about a PBES
  - *pbes2pp*: Pretty prints a PBES
- **Solving:**
  - *pbes2bool*: Solves a PBES
- **Optimisation:**
  - *pbesrewr*: Rewrite data expressions in a PBES
Toolset overview
Import and export

Import and export tools:
- **chi2mcrl2**: Translates a $\chi$ specification to an mCRL2 specification
- **pnml2mcrl2**: Translates a Petri net to an mCRL2 specification
- **tbf2lps**: Translates a $\mu$CRL LPE to an mCRL2 LPS
- **formcheck**: Checks whether a boolean data expression holds
- **lps2torx**: Provide TorX explorer interface to an LPS
Toolset overview

Tools under development

Graphical specification (individual component):
Toolset overview
Tools under development

Graphical specification (communicating components):
Toolset overview

Tools under development (2)

Systems Quality Analysis and Design Toolkit (SQuADT):
Toolset demo: dining philosophers

Dining philosophers:

1. Problem description
2. Model the problem
3. Verify the problem
4. A solution
5. Verify the solution
Toolset demo: dining philosophers

Problem description

- Illustrative example of a common computing problem in concurrency
- 5 hungry philosophers
- 5 forks in-between the philosophers
- Rules:
  - Philosophers cannot communicate
  - Two forks are needed for eating
Toolset demo: dining philosophers

Problem description (2)

- **Deadlock**: Every philosopher holds a left fork and waits for a right fork (or vice versa).
- **Starvation**: If a philosopher cannot acquire two forks he will starve.

The dining philosophers problem is a generic and abstract problem used for explaining various issues which arise in concurrency theory.

- The forks resemble shared resources.
- The philosophers resemble concurrent processes.
Toolset demo: dining philosophers
Modelling the problem: data types

Data type for representing the philosophers and the forks:

\[
\begin{align*}
\text{sort} & \quad \text{PhilId} = \text{struct } p_1 &|& p_2 &|& p_3 &|& p_4 &|& p_5; \\
& \quad \text{ForkId} = \text{struct } f_1 &|& f_2 &|& f_3 &|& f_4 &|& f_5;
\end{align*}
\]

Function for representing the positions of the forks relative to the philosophers (the left and right fork):

\[
\begin{align*}
\text{map} & \quad \text{lf, rf} : \text{PhilId} \rightarrow \text{ForkId}; \\
\text{eqn} & \quad \text{lf}(p_1) = f_1; \quad \text{lf}(p_2) = f_2; \quad \text{lf}(p_3) = f_3; \\
& \quad \text{lf}(p_4) = f_4; \quad \text{lf}(p_5) = f_5; \\
& \quad \text{rf}(p_1) = f_5; \quad \text{rf}(p_2) = f_1; \quad \text{rf}(p_3) = f_2; \\
& \quad \text{rf}(p_4) = f_3; \quad \text{rf}(p_5) = f_4;
\end{align*}
\]
Toolset demo: dining philosophers

Modelling the problem: individual processes

Modelling the behaviour of the philosophers:

- **eat**\((p)\): philosopher \(p\) eats
- **get**\((p, f)\): philosopher \(p\) takes up fork \(f\)
- **put**\((p, f)\): philosopher \(p\) puts down fork \(f\)

\[
\begin{align*}
\text{act} & \quad \text{get, put : PhilId } \times \text{ForkId}; \\
& \quad \text{eat : PhilId}; \\
\text{proc} & \quad \text{Phil}(p : \text{PhilId}) = \\
& \quad \quad (\text{get}(p, \text{lf}(p)) \cdot \text{get}(p, \text{rf}(p)) + \text{get}(p, \text{rf}(p)) \cdot \text{get}(p, \text{lf}(p))) \\
& \quad \quad \cdot \text{eat}(p) \\
& \quad \quad \cdot (\text{put}(p, \text{lf}(p)) \cdot \text{put}(p, \text{rf}(p)) + \text{put}(p, \text{rf}(p)) \cdot \text{put}(p, \text{lf}(p))) \\
& \quad \quad \cdot \text{Phil}(p);
\end{align*}
\]
Toolset demo: dining philosophers
Modelling the problem: individual processes

Modelling the behaviour of the forks:

- \text{up}(p, f): \text{fork } f \text{ is picked up by philosopher } p
- \text{down}(p, f): \text{fork } f \text{ is put down by philosopher } p

\[
\begin{align*}
\text{act} & \quad \text{up, down} : \text{PhilId} \times \text{ForkId}; \\
\text{proc} & \quad \text{Fork}(f : \text{ForkId}) = \\
& \quad \sum_{p : \text{Phil}} \text{up}(p, f) \cdot \text{down}(p, f) \cdot \text{Fork}(f);
\end{align*}
\]
Toolset demo: dining philosophers
Modelling the problem: communication and initialisation

Complete specification:
- put all forks and philosophers in parallel
- synchronise on actions get and up, and on actions put and down

\[
\text{act} \quad \text{lock, free} : \text{PhilId} \times \text{ForkId}; \\
\text{init} \quad \nabla (\{\text{lock, free, eat}\}, \\
\quad \Gamma (\{\text{get|up} \rightarrow \text{lock, put|down} \rightarrow \text{free}\}, \\
\quad \text{Phil}(p_1) \parallel \text{Phil}(p_2) \parallel \text{Phil}(p_3) \parallel \text{Phil}(p_4) \parallel \text{Phil}(p_5) \parallel \\
\quad \text{Fork}(f_1) \parallel \text{Fork}(f_2) \parallel \text{Fork}(f_3)) \parallel \text{Fork}(f_4) \parallel \text{Fork}(f_5));
\]
Toolset demo: dining philosophers
Analysing the model

- Linearisation:
  \texttt{mcrl22lps -vD dining5.mcrl2 dining5.lps}

- Sum instantiation:
  \texttt{lpssuminst -v dining5.lps dining5.sum.lps}

- Constant elimination:
  \texttt{lpsconstelm -v dining5.sum.lps dining5.sum.const.lps}

- Parameter elimination:
  \texttt{lpsparelm -v dining5.sum.const.lps dining5.sum.const.par.lps}

- Generate state space:
  \texttt{lps2lts -vD dining5.sum.const.lps dining5.sum.const.lts}

- Deadlock detected!
Toolset demo: dining philosophers

A Possible solution: the waiter

Waiter:
- Decides whether a philosopher may pick up two forks
- Only allowed when less than four forks are in use
Toolset demo: dining philosophers

Modelling the solution: actions

New actions:

- \text{ack}(p)\colon \text{philosopher } p \text{ takes the opportunity to pick up two forks and eat}
- \text{done}(p)\colon \text{philosopher } p \text{ signal the waither that he is done eating and has put down both forks}

\begin{center}
\begin{verbatim}
act r_\text{ack}, s_\text{ack}, \text{ack} : \text{Phil};
       r_\text{done}, s_\text{done}, \text{done} : \text{Phil};
\end{verbatim}
\end{center}
Toolset demo: dining philosophers
Modelling the solution: the waiter

Modelling the behaviour of the waiter:

\[
\text{proc } \text{Waiter}(n : \mathbb{N}) = \\
(n < 4) \rightarrow \sum_{p:Phil} \text{s-ack}(p) \cdot \text{Waiter}(n+2) \\
+ (n > 1) \rightarrow \sum_{p:Phil} \text{r-done}(p) \cdot \text{Waiter}(\text{Int2Nat}(n-2));
\]
Toolset demo: dining philosophers

Modelling the solution: the philosophers

Extend the philosopher process:

\[
\text{proc } \text{Phil}(p : \text{PhilId}) = \\
\quad \text{r\_ack}(p) \\
\quad \cdot (\text{get}(p, \text{lf}(p)) \cdot \text{get}(p, \text{rf}(p)) + \text{get}(p, \text{rf}(p)) \cdot \text{get}(p, \text{lf}(p))) \\
\quad \cdot \text{eat}(p) \\
\quad \cdot (\text{put}(p, \text{lf}(p)) \cdot \text{put}(p, \text{rf}(p)) + \text{put}(p, \text{rf}(p)) \cdot \text{put}(p, \text{lf}(p))) \\
\quad \cdot \text{s\_done}(p) \\
\quad \cdot \text{Phil}(p);
\]
Toolset demo: dining philosophers
Modelling the solution: communication and initialisation

Complete specification:

\[
\text{init} \quad \nabla \{ \text{lock, free, eat, ack, done}\}, \\
\Gamma \{ \text{get|up } \rightarrow \text{lock, put|down } \rightarrow \text{free} \\
\text{r_ack|s_ack } \rightarrow \text{ack, r_done|s_done } \rightarrow \text{done,} \\
\text{Phil}(p_1) \parallel \text{Phil}(p_2) \parallel \text{Phil}(p_3) \parallel \text{Phil}(p_4) \parallel \text{Phil}(p_5) \parallel \\
\text{Fork}(f_1) \parallel \text{Fork}(f_2) \parallel \text{Fork}(f_3) \parallel \text{Fork}(f_4) \parallel \text{Fork}(f_5) \parallel \\
\text{Waiter}(0) \\
\} 
\]
**Toolset demo: dining philosophers**

**Verifying the solution**

- **Deadlock freedom:** Yes

  \[[true^*]⟨true\rangle \text{true}\]

  1. `lps2pbes --formula=nodeadlock.mcf dining5_waiter.lps dining5_waiter_nd.pbes`
  2. `pbes2bool dining5_waiter_nd.pbes`

- **Starvation freedom:** Yes

  \[\forall_{p:Phil} [true^* \cdot (¬\text{eat}(p))^*] \langle(¬\text{eat}(p))^* \cdot \text{eat}(p)\rangle \text{true}\]

  1. `lps2pbes --formula=nostarvation.mcf dining5_waiter.lps dining5_waiter_ns.pbes`
  2. `pbes2bool dining5_waiter_ns.pbes`
Industrial case studies

Industrial case studies carried out using the $\mu$CRL and mCRL2 toolsets:

- Océ: automated document feeder
- Add-controls: distributed system for lifting trucks
- CVSS: automated parking garage
- Vitatron: pacemaker
- AIA: ITP load-balancer
- Philips Healthcare: patient support platform
- ... and lots more
Industrial case studies
Océ: automated document feeder

Automated document feeder:
- Feed documents to the scanner automatically
- One sheet at a time
- Prototype implementation

Analysis:
- Model: $\mu$CRL
- Verification: CADP
- Size: 350,000 states and 1,100,000 transitions
- Actual errors found: 2
Industrial case studies
Add-controls: distributed system for lifting trucks

Distributed system for lifting trucks:
- Each lift has a controller
- Controllers are connected via a circular network
- 3 errors found after testing by the developers

Analysis:
- Model: $\mu$CRL
- Verification: CADP
- Actual errors found: 4

<table>
<thead>
<tr>
<th>Lifts</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>383</td>
<td>716</td>
</tr>
<tr>
<td>3</td>
<td>7,282</td>
<td>18,957</td>
</tr>
<tr>
<td>4</td>
<td>128,901</td>
<td>419,108</td>
</tr>
<tr>
<td>5</td>
<td>2,155,576</td>
<td>8,676,815</td>
</tr>
</tbody>
</table>
Industrial case studies
CVSS: automated parking garage

An automated parking garage:
Industrial case studies
CVSS: automated parking garage (2)

Verified design of an automated parking garage:
- Design of the control software
- Verified the safety layer of this design

Analysis:
- Design: 991 lines of mCRL2
- Verification: 217 lines of mCRL2
- Size: 3.3 million states and 98 million transitions
- Simulation using custom built visualization plugin
Industrial case studies
CVSS: automated parking garage (3)

Design flaws detected using the visualisation plugin:

a

b
Industrial case studies
Vitatron: pacemaker

Pacemaker:
- Controlled by firmware
- Must deal with all possible rates and arrhythmias
- Firmware design

Analysis:
- Model: mCRL2 (and Uppaal)
- Verification: mCRL2 model checking
- Size:
  - full model: 500 million states
  - suspicious part: 714,464 states
- Actual errors found: 1 (known)
Industrial case studies
AIA: ITP load balancer

Intelligent Text Processing (ITP):
- Print job distribution over document processors
- 7,500 lines of C code

Analysis:
- Load balancing part
- Model: mCRL2
- Verification:
  - mCRL2 model checking
- Actual errors found: 6
- Size: 1.9 billion states and 38.9 billion transitions
- LaQuSo certification
Industrial case studies
Philips Healthcare: patient support platform

Patient support platform:
- Verified design of the control software
- Convertor and Motion Controller
- Implemented in Python

Analysis:
- Model: mCRL2
- Verification: CADP
- Requirements:
  - 4 checked
  - 1 did not hold but was very unlikely to occur
- Size: 45 million states