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:

$43,252,003,274,489,856,000$ (4.3×10^{19}) 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 (μCRL)
- During 1990s: development of the μCRL toolset
- Since 2004: micro Common Representation Language 2 (mCRL2) + toolset

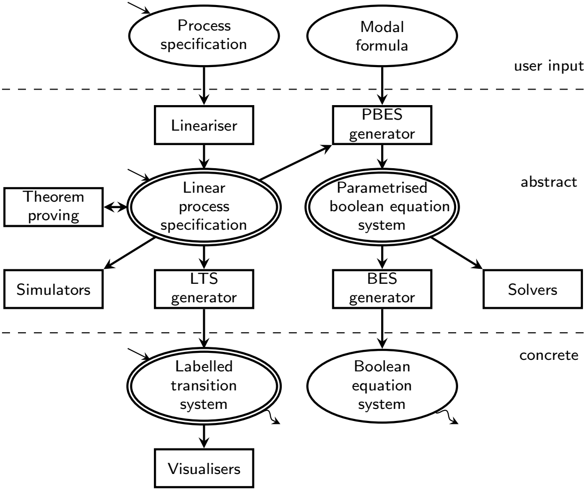
Toolset overview

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



Toolset overview

Linear process specifications

LPS tools:

- Generation:
  - mcrl22lps: 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)

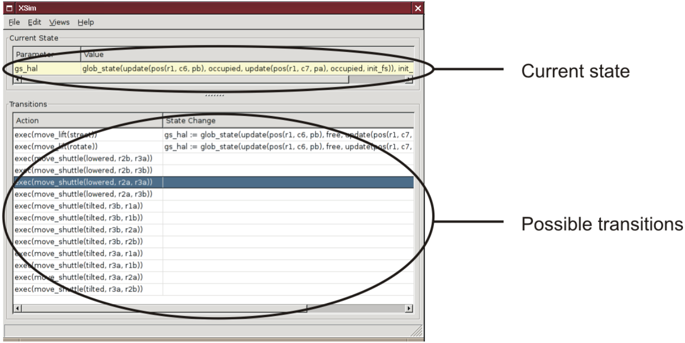
LPS tools:

- 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)

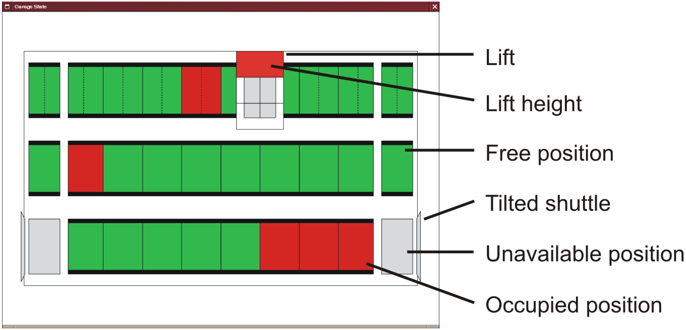
Simulation using xsim:



Toolset overview

Linear process specifications (3)

Simulation using xsim with plugins:



Toolset overview

Labelled transition systems

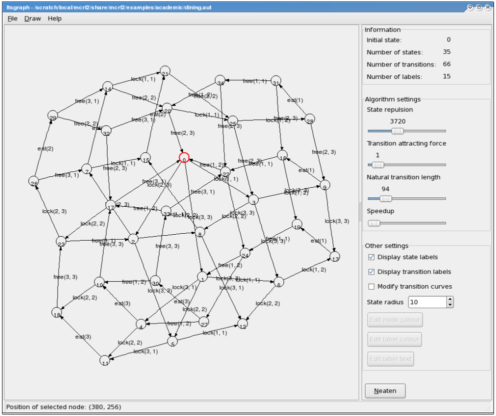
LTS tools:

- 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)

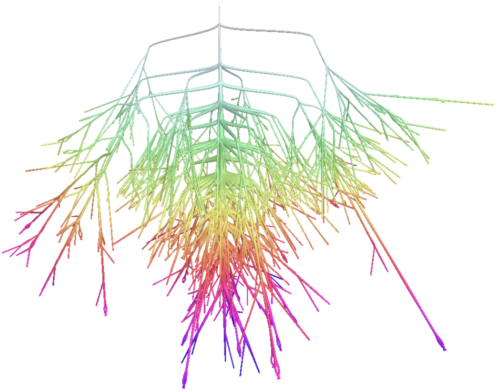
Visualisation using ltsgraph:



Toolset overview

Labelled transition systems (3)

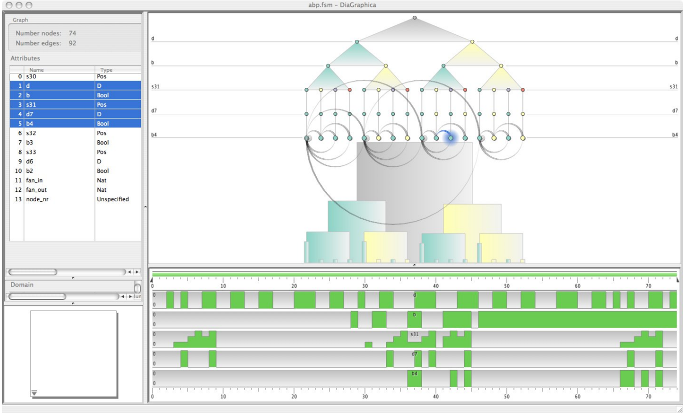
Visualisation using ltsview:



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

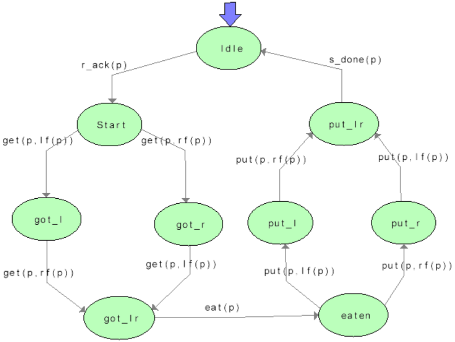
Import and export tools:

- chi2mcrl2: Translates a χ specification to an mCRL2 specification
- pnml2mcrl2: Translates a Petri net to an mCRL2 specification
- tbf2lps: Translates a μCRL 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

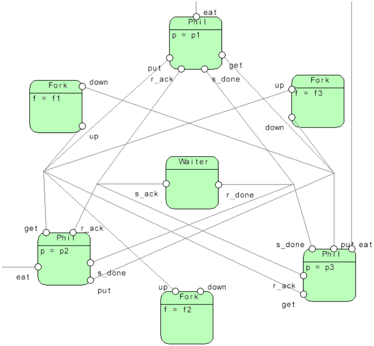
Graphical specification (individual component):



Toolset overview

Tools under development

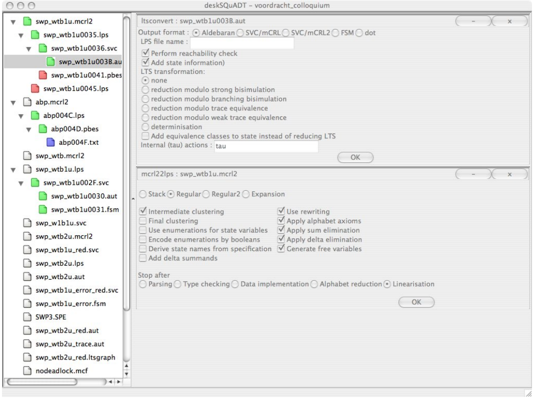
Graphical specification (communicating components):



Toolset overview

Tools under development (2)

Systems Quality Analysis and Design Toolkit (SQuADT):



Toolset demo: dining philosophers

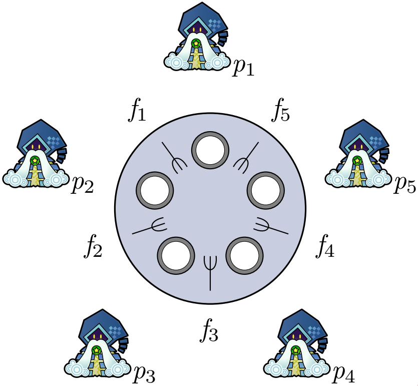
Dining philosophers:

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:

$$\textbf{sort} \quad \mathit{PhilId} = \textbf{struct}\ p_1 \mid p_2 \mid p_3 \mid p_4 \mid p_5;$$
$$\mathit{ForkId} = \textbf{struct}\ f_1 \mid f_2 \mid f_3 \mid f_4 \mid f_5;$$

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

$$\textbf{map} \quad \mathit{lf}, \mathit{rf} : \mathit{PhilId} \to \mathit{ForkId};$$
$$\textbf{eqn} \quad \mathit{lf}(p_1) = f_1;\ \mathit{lf}(p_2) = f_2;\ \mathit{lf}(p_3) = f_3;$$
$$\mathit{lf}(p_4) = f_4;\ \mathit{lf}(p_5) = f_5;$$
$$\mathit{rf}(p_1) = f_5;\ \mathit{rf}(p_2) = f_1;\ \mathit{rf}(p_3) = f_2;$$
$$\mathit{rf}(p_4) = f_3;\ \mathit{rf}(p_5) = f_4;$$

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

$$
\begin{array}{ll}
\textbf{act} & \mathsf{get}, \mathsf{put} : \mathit{PhilId} \times \mathit{ForkId}; \\
 & \mathsf{eat} : \mathit{PhilId}; \\
\textbf{proc} & \mathsf{Phil}(p : \mathit{PhilId}) = \\
 & \quad (\mathsf{get}(p, \mathit{lf}(p)) \cdot \mathsf{get}(p, \mathit{rf}(p)) + \mathsf{get}(p, \mathit{rf}(p)) \cdot \mathsf{get}(p, \mathit{lf}(p))) \\
 & \quad \cdot\, \mathsf{eat}(p) \\
 & \quad \cdot\, (\mathsf{put}(p, \mathit{lf}(p)) \cdot \mathsf{put}(p, \mathit{rf}(p)) + \mathsf{put}(p, \mathit{rf}(p)) \cdot \mathsf{put}(p, \mathit{lf}(p))) \\
 & \quad \cdot\, \mathsf{Phil}(p);
\end{array}
$$

Toolset demo: dining philosophers

Modelling the problem: individual processes

Modelling the behaviour of the forks:

- up(p, f): fork f is picked up by philosopher p
- down(p, f): fork f is put down by philosopher p

$$\begin{array}{ll} \textbf{act} & \mathsf{up}, \mathsf{down} : \mathit{PhilId} \times \mathit{ForkId}; \\ \textbf{proc} & \mathsf{Fork}(f : \mathit{ForkId}) = \\ & \qquad \sum_{p:\mathit{Phil}} \mathsf{up}(p, f) \cdot \mathsf{down}(p, f) \cdot \mathsf{Fork}(f); \end{array}$$

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

$$
\begin{array}{ll}
\textbf{act} & \mathsf{lock}, \mathsf{free} : \mathit{PhilId} \times \mathit{ForkId}; \\
\textbf{init} & \nabla(\{\mathsf{lock}, \mathsf{free}, \mathsf{eat}\}, \\
& \quad \Gamma(\{\mathsf{get}|\mathsf{up} \rightarrow \mathsf{lock}, \mathsf{put}|\mathsf{down} \rightarrow \mathsf{free}\}, \\
& \qquad \mathsf{Phil}(p_1) \parallel \mathsf{Phil}(p_2) \parallel \mathsf{Phil}(p_3) \parallel \mathsf{Phil}(p_4) \parallel \mathsf{Phil}(p_5) \parallel \\
& \qquad \mathsf{Fork}(f_1) \parallel \mathsf{Fork}(f_2) \parallel \mathsf{Fork}(f_3))) \parallel \mathsf{Fork}(f_4) \parallel \mathsf{Fork}(f_5) \\
& \qquad));
\end{array}
$$

Toolset demo: dining philosophers

Analysing the model

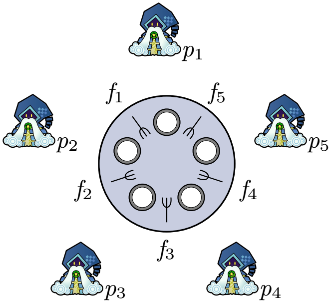
- Linearisation:
  mcrl22lps -vD dining5.mcrl2 dining5.lps
- Sum instantation:
  lpssuminst -v dining5.lps dining5.sum.lps
- Constant elimination:
  lpsconstelm -v dining5.sum.lps dining5.sum.const.lps
- Parameter elimination:
  lpsparelm -v dining5.sum.const.lps
  dining5.sum.const.par.lps
- Generate state space:
  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:

- ack(p): philosopher p takes the opportunity to pick up two forks and eat
- done(p): philospher p signal the waither that he is done eating and has put down both forks

act	r_ack, s_ack, ack : $Phil$;
	r_done, s_done, done : $Phil$;

Toolset demo: dining philosophers

Modelling the solution: the waiter

Modelling the behaviour of the waiter:

$$
\begin{array}{ll}
\textbf{proc} & \mathsf{Waiter}(n:\mathbb{N}) = \\
& \quad (n<4) \rightarrow \sum_{p:Phil} \mathsf{s\_ack}(p)\cdot \mathsf{Waiter}(n{+}2) \\
& + \ (n>1) \rightarrow \sum_{p:Phil} \mathsf{r\_done}(p)\cdot \mathsf{Waiter}(Int2Nat(n{-}2));
\end{array}
$$

Toolset demo: dining philosophers

Modelling the solution: the philosophers

Extend the philosopher process:

$$
\begin{aligned}
\textbf{proc}\quad & \mathsf{Phil}(p : \mathit{PhilId}) = \\
& \;\mathsf{r\_ack}(p) \\
& \cdot (\mathsf{get}(p, \mathit{lf}(p)) \cdot \mathsf{get}(p, \mathit{rf}(p)) + \mathsf{get}(p, \mathit{rf}(p)) \cdot \mathsf{get}(p, \mathit{lf}(p))) \\
& \cdot \mathsf{eat}(p) \\
& \cdot (\mathsf{put}(p, \mathit{lf}(p)) \cdot \mathsf{put}(p, \mathit{rf}(p)) + \mathsf{put}(p, \mathit{rf}(p)) \cdot \mathsf{put}(p, \mathit{lf}(p))) \\
& \cdot \mathsf{s\_done}(p) \\
& \cdot \mathsf{Phil}(p);
\end{aligned}
$$

Toolset demo: dining philosophers

Modelling the solution: communication and initialisation

Complete specification:

init $\nabla(\{\text{lock}, \text{free}, \text{eat}, \text{ack}, \text{done}\},$
$\Gamma(\{\text{get}|\text{up} \rightarrow \text{lock}, \text{put}|\text{down} \rightarrow \text{free}$
$\text{r\_ack}|\text{s\_ack} \rightarrow \text{ack}, \text{r\_done}|\text{s\_done} \rightarrow \text{done},$
$\text{Phil}(p_1) \parallel \text{Phil}(p_2) \parallel \text{Phil}(p_3) \parallel \text{Phil}(p_4) \parallel \text{Phil}(p_5) \parallel$
$\text{Fork}(f_1) \parallel \text{Fork}(f_2) \parallel \text{Fork}(f_3) \parallel \text{Fork}(f_4) \parallel \text{Fork}(f_5) \parallel$
$\text{Waiter}(0)$
$));$

Toolset demo: dining philosophers

Verifying the solution

- Deadlock freedom: Yes

$$[true^*]\,\langle true\rangle\, true$$

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

- Starvation freedom: Yes

$$\forall_{p:Phil}\,[true^* \cdot (\neg \mathsf{eat}(p))^*]\,\langle(\neg \mathsf{eat}(p))^* \cdot \mathsf{eat}(p)\rangle\, true$$

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 μCRL 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

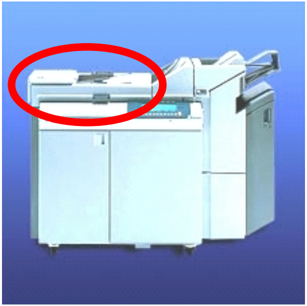
Océ: automated document feeder

Automated document feeder:

- Feed documents to the scanner automatically
- One sheet at a time
- Prototype implementation

Analysis:

- Model: μCRL
- 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: μCRL
- Verification: CADP
- Actual errors found: 4

Lifts	States	Transitions
2	383	716
3	7,282	18,957
4	128,901	419,108
5	2,155,576	8,676,815

Industrial case studies

CVSS: automated parking garage

An automated parking garage:

Industrial case studies

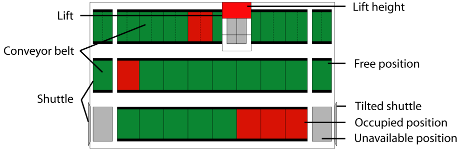
CVSS: automated parking garage (2)

Verified design of an automated parking garage:

- Design of the control software
- Verified the safety layer of this design

Analysis:

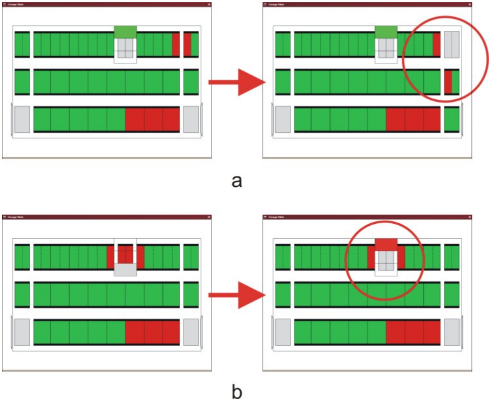
- Design: 991 lines of mCRL2
- Verification: 217 lines of mCRL2
- Size: 3.3 million states and 98 million transitions
- Simulation using custom built visualisation plugin



Industrial case studies

CVSS: automated parking garage (3)

Design flaws detected using the visualisation plugin:



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

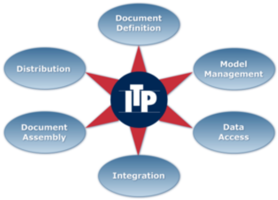
AIA: ITP load balancer

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