mCRL2

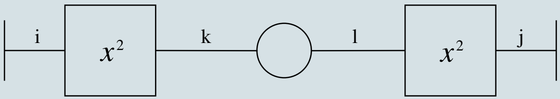
Towards a practical formal specification language

Aad Mathijssen Jan Friso Groote Muck van Weerdenburg Yaroslav Usenko

23th June 2005

Motivation: Petri nets

Bring stand-alone developments of specification languages together.
GenSpect: find a common base for hierarchical Petri nets and process algebra with data.



It should be possible to translate Petri nets to process algebra:

- places are unordered buffers
- transitions are memoryless input/output relations
- arcs define communication between places and transitions

Motivation: Petri nets (2)

We would like to use μCRL as a target for this translation. Unfortunately, there are a number of problems:

- all actions involved in the firing of transitions occur at the same time; using interleaving for this translation is problematic:
  - state space explosion
  - nice Petri net properties do not carry over
- hierarchical approach enforces that operators are compositional, but communication is not

Motivation: concrete data types

Problems with the use of μCRL in practise, because of the lack of *concrete data types*:

- specifications are too long
- specifications are hard to read
- standard notions are specified differently amongst different specifications
- lack of higher-order notions

Specifying all data types yourself distracts from doing the real work.

Motivation: linear process equations

Every guarded untimed μCRL specification can be transformed to a linear process equation (LPE), which has the following form:

$$P(\overrightarrow{d{:}D}) = \sum_{i \in I} \sum_{\overrightarrow{e_i{:}E_i}} a_i(\overrightarrow{f_i}(\overrightarrow{d, e_i})) \cdot P(\overrightarrow{g_i}(\overrightarrow{d, e_i})) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta$$

An LPE is a symbolic representation of a state space.
It is the core language used by the μCRL toolset.

Two things are lacking:

- time
- don't care values

mCRL2

Design a new language and toolset, using both theoretical and practical experience with μCRL. Basically, the mCRL2 language is timed μCRL with the following changes/additions:

- true concurrency (multi-actions)
- local communication
- higher-order algebraic specification
- concrete data types

The toolset will use a new LPE format, which supports multi-actions, higher-order algebraic specification, time and don't care values.

mCRL2 (2)

To find out if the language and the toolset is useful in *practise*, we took the following approach to design the language:

1. start with an initial design of the language and a toolset
2. iteratively:
   (a) test using real-world examples
   (b) improve formal language
   (c) improve toolset

mCRL2 process language

Process expressions have the following syntax:

$$
\begin{array}{rl}
p ::= & a(\vec{d}) \mid \delta \mid \tau \mid p+p \mid p\cdot p \mid p \parallel p \mid p \mathbin{⌊\!⌊} p \mid p|p \mid X(\vec{d}) \\
& \mid (d=d) \rightarrow p, p \mid p \cdot d \mid \sum_{\vec{x}:\vec{s}} p \\
& \mid \nabla_V(p) \mid \partial_{IH}(p) \mid \tau_{IH}(p) \mid \Gamma_C(p) \mid \rho_R(p)
\end{array}
$$

- sync operator $|$ does not communicate
- a sync of actions is called a *multi-action*, e.g. $a,\ a|b,\ b|a,\ a|b|c,\ a|b|a,\ a(t)|b(u)|a(v)$
- V and IH are sets of parameterless multi-actions/actions
- C and R are sets of renamings of parameterless multi-actions/actions to actions; the lhs's of C/R must be disjoint

mCRL2 process language (2)

Restriction and communication:

- allow operator ∇_V *only* multi-actions that are in the set V, e.g.
  $\nabla_{\{a,b\}}(a \parallel b) = a \cdot b + b \cdot a,\ \nabla_{\{a|b\}}(a \parallel b) = a|b,$
  $\nabla_{\{a|b\}}(a|b|c) = \delta,\ \nabla_{\{a,b|c\}}(a \parallel b \parallel c) = a \cdot (b|c) + (b|c) \cdot a$
- blocking operator ∂_{IH} blocks all actions that occur in the set IH, e.g.
  $\partial_{\{a\}}(a + b \cdot (a|c)) = b \cdot \delta$
- communication operator Γ_C realises communication of multi-actions with equal parameters, e.g. where $t = u$ and $t \neq v$:
  $\Gamma_{\{a|b \to c\}}(a(t)|b(u)) = c(t),\ \Gamma_{\{a|b \to c\}}(a(t)|b(v)) = a(t)|b(v),$
  $\Gamma_{\{a|b|c \to d\}}(a|b|c|d) = d|d,\ \Gamma_{\{a|b|c \to d, d|d \to d\}}(a|b|c|d) = d|d$
  $\sum_{d:D} \Gamma_{\{a|a \to a\}}(a(d)|a(t)) = \sum_{d:D} d = t \to a(t), a(d)|a(t)$

mCRL2 process language (3)

Process equations are formed as follows:

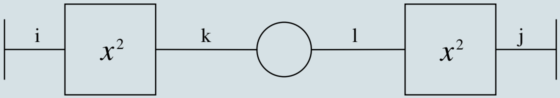
$$pe ::= X(\overrightarrow{x : s}) = p$$

Process specifications:

$$sp ::= (\textbf{act}\ (a;\ |\ a : s \times \cdots \times s;\ )^{+}\ |\ \textbf{proc}\ (pe;\ )^{+})^{*}\ \textbf{init}\ p;$$

Petri net translation

Petri nets can be expressed in mCRL2:



Translation to mCRL2:

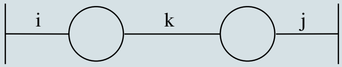
$$
\begin{array}{ll}
Sqr_{i,o} & = \sum_{n:\mathbb{N}} \overline{get_i}(n) | \overline{put_o}(n^2) \cdot Sqr_{i,o} \\
P_{i,o}(b : Bag(\mathbb{N})) & = \sum_{n:\mathbb{N}} \underline{put_i}(n) \cdot P_{i,o}(b \cup \{ n \}) + \\
 & \sum_{n:\mathbb{N}} n \in b \rightarrow \underline{get_o}(n) \cdot P_{i,o}(b \setminus \{ n \}) \\
DSqr_{i,j} & = \nabla_V(\Gamma_C(Sqr_{i,k} \parallel \overline{P_{k,l}(\emptyset)} \parallel Sqr_{l,j}))
\end{array}
$$

where

$$C = \{ \overline{put_k} | \underline{put_k} \rightarrow put_k, \overline{get_l} | \underline{get_l} \rightarrow get_l \}, V = \{ \overline{get_i} | put_k, get_l | \overline{put_j} \}$$

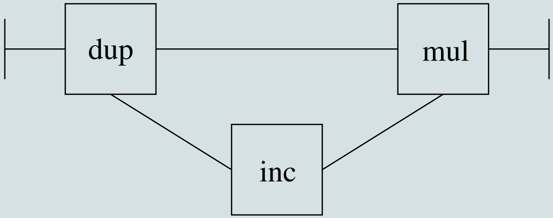
Beyond Petri nets

Connected places:



$$P^2 = \nabla_{\{\underline{put_i}, pass_k, \underline{get_j}\}}(\Gamma_{\{\underline{get_k}|\underline{get_k} \rightarrow pass_k\}}(P_{i,k}(\emptyset) \parallel P_{k,j}(\emptyset)))$$

Connected transitions:



mCRL2 data language

Is it advantageous to use an existing data language?
Not likely, because:

- algebraic specification languages are often *first-order* and lack *concrete data types*
- functional programming languages cannot handle *open terms* and are focused on *evaluation* only
- it is often hard to integrate an existing language in a toolset

mCRL2 data language (2)

Conclusion: we define our own language, but keep the door open to existing algebraic specification languages.

Approach:

- define a core theory of higher-order algebraic specification
- add concrete data types:
  - add syntax
  - implement data types within the core theory

Higher-order algebraic specification

Concepts: sorts, operations, terms and equations

Higher-order *sorts* are constructed as follows, where b is a set of *base* sorts:

$$s := b \mid s \to s$$

An *operation* is of the form $f : s$, which means that all operations are constants.

Data *terms* are constructed from variables and operations:

$$d ::= x : s \mid f : s \mid d(d)$$

Higher-order algebraic specification (2)

We use a *conditional equational logic* to express properties of data:

$$\phi ::= \forall \overrightarrow{x{:}s}.\, d = d \ \wedge \cdots \wedge \ d = d \rightarrow d = d$$

Data specification elements:

$$\begin{aligned} dse ::= {} & \mathbf{sort}\ (b;)^{+} \\ & \mid \mathbf{cons}\ (f : s;)^{+} \\ & \mid \mathbf{map}\ (f : s;)^{+} \\ & \mid (\mathbf{var}\ (x : s;)^{+})?\ \mathbf{eqn}\ (\phi;)^{+} \end{aligned}$$

Data specification:

$$ds ::= dse^{*}$$

HOAS in practise

Changes/additions:

- conditional equations are restricted to $d \rightarrow d = d$, where the condition is a term of predefined sort $\mathbb{B}$
- $s_0 \times \cdots \times s_n \rightarrow s$ is a shorthand for $s_0 \rightarrow \cdots \rightarrow s_n \rightarrow s$, where $\rightarrow$ is right-associative
- $t(t_0, \ldots, t_n)$ is a shorthand for $t(t_0) \cdots (t_n)$, where application is left-associative
- sort references can be defined:

$$\textbf{sort } B = C \rightarrow D;$$

- add prefix, infix and mixfix notation for concrete data types, together with operator precedence

HOAS in practise: concrete data types

General:

- equality $d == d$, inequality $d \neq d$ and conditional $\mathit{if}(d, d, d)$
- lambda expressions $\lambda\overrightarrow{x{:}s}.d$
- where clauses d **whr** $x = d, \ldots, x = d$ **end**

Basic data types:

- Booleans ($\mathbb{B}$)
  $\mathit{true}, \mathit{false}, \neg d, d \wedge d, d \vee d, d \Rightarrow d, \forall\overrightarrow{x{:}s}.d, \exists\overrightarrow{x{:}s}.d$
- Numbers ($\mathbb{P}$, $\mathbb{N}$ and $\mathbb{Z}$)
  $0, 1, -1, 2, -2, \ldots$
  $d < d, d \leq d, d > d, d \geq d, -d, d + d, d - d, d * d, d\ \mathbf{div}\ d, d\ \mathbf{mod}\ d, \ldots$

HOAS in practise: concrete data types (2)

Type constructors:

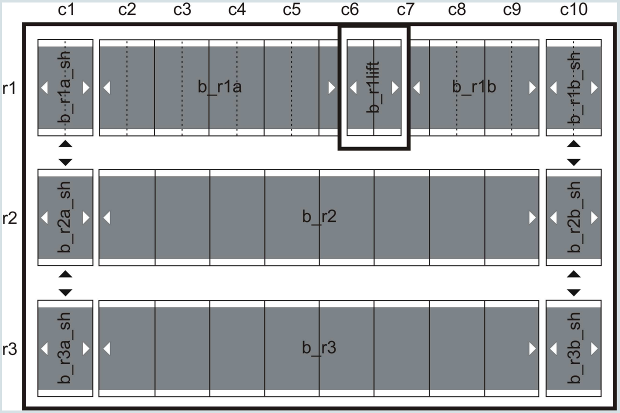
- structured types (sum types and product types)

$$
\begin{array}{rl}
\textbf{struct} & c_1(pr_{1,1} : A_{1,1}, \ \ldots, pr_{1,k_1} : A_{1,k_1})?is\_c_1 \\
 & \mid c_2(pr_{2,1} : A_{2,1}, \ \ldots, pr_{2,k_2} : A_{2,k_2})?is\_c_2 \\
 & \vdots \\
 & \mid c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?is\_c_n
\end{array}
$$

- lists ($List(s)$)
  $[\,], [d, \ldots, d], \#d, d \triangleright d, d \triangleleft d, d +\!\!+ d, d.d$
- sets and bags ($Set(s)$, $Bag(s)$)
  $\emptyset, \{\, d, \ldots, d \,\}, \{\, d{:}d, \ldots, d{:}d \,\}, \{\, x{:}s \mid d \,\}$
  $\#d, d \in d, d \subseteq d, d \subset d, d \cup d, d \setminus d, d \cap d, \overline{d}$

Example: automated parking garage

The company CVSS is currently building an automated parking garage in Bremen.



LaQuSo assignment: design and analyse the software for this system.

Focus on safety.

Implementation of concrete data types

General requirements:

- computability: reading the equations from left to right, we obtain a term rewrite system that is confluent, terminating and complete (if possible)
- simplicity: internal representation should be unique
- efficiency:
  - reduction lengths should be minimised
  - the number of equations should be minimised
- provability: the number of properties that can be proved on open terms should be maximised

Implementation of concrete data types (2)

Data type specific:

- lambda expressions and where clauses are implemented as named functions, e.g. $\lambda y{:}\mathbb{N}.(x+y)$ becomes $f(x)$, where $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ satisfies $f(x)(y) = x + y$, for all $x, y : \mathbb{N}$
- quantifications over sort s are implemented as functions of sort $(s \to \mathbb{B}) \to \mathbb{B}$
- numbers have a unique binary representation:
  - sort $\mathbb{P}$ has constructors $1 : \mathbb{P}$ and $cDub : \mathbb{B} \times \mathbb{P} \to \mathbb{P}$
  - sort $\mathbb{N}$ has constructors $0 : \mathbb{N}$ and $cNat : \mathbb{P} \to \mathbb{N}$
  - sort $\mathbb{Z}$ has constructors $cInt : \mathbb{N} \to \mathbb{Z}$ and $cNeg : \mathbb{P} \to \mathbb{Z}$
- sets and bags over sort s are implemented as functions $s \to \mathbb{B}$ and $s \to \mathbb{N}$

Linear process equations

μCRL LPE:

$$P(\overrightarrow{d{:}D}) = \sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} a_i(\overrightarrow{f_i}(\overrightarrow{d,e_i})) \cdot P(\overrightarrow{g_i}(\overrightarrow{d,e_i})) \lhd c_i(\overrightarrow{d,e_i}) \rhd \delta$$

mCRL2 LPE:

$$\begin{aligned} P(\overrightarrow{d{:}D}) = \sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} & c_i(\overrightarrow{d,e_i}) \rightarrow \\ & (a_i^0(\overrightarrow{f_{i,0}}(\overrightarrow{d,e_i})) \mid \cdots \mid a_i^{n(i)}(\overrightarrow{f_{i,n(i)}}(\overrightarrow{d,e_i}))) \cdot t_i(\overrightarrow{d,e_i}) \cdot P(\overrightarrow{g_i}(\overrightarrow{d,e_i})), \end{aligned}$$

where:

- data types are higher-order
- *free* variables are used to model don't care values

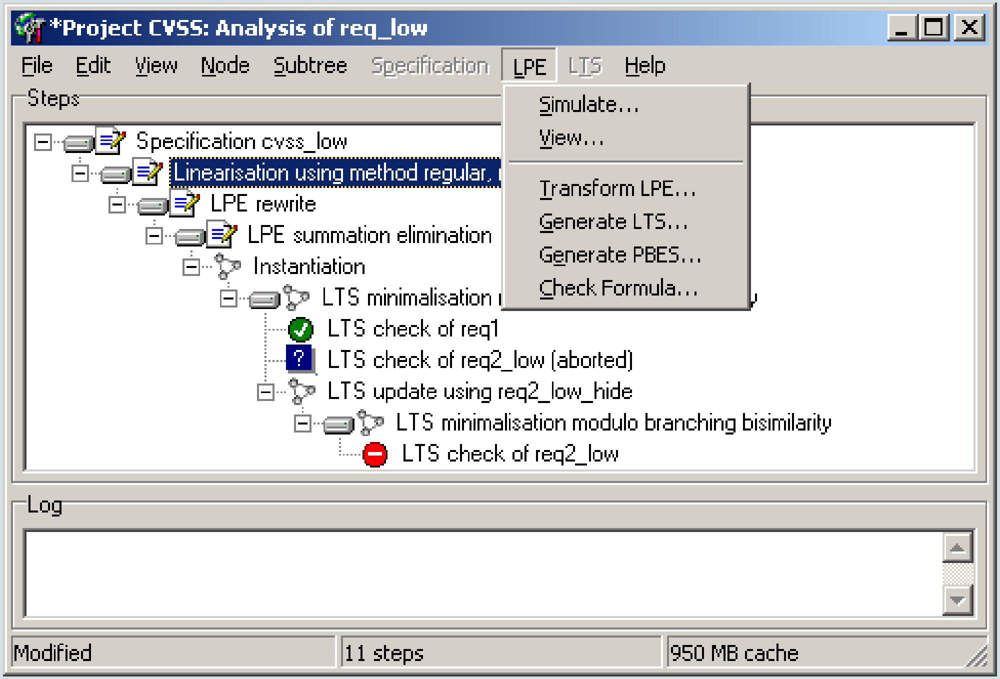
Tool support

Because of the changes to the core language (LPEs), reuse of existing tools is hard. So we re-implemented some of them.

New goals:

- graphical user interface that will:
  - lower the treshold for new users
  - simplify the analysis process
- flexible LPE simulator with different pluggable views
- model checking directly on LPEs
- visualisation of large LTSs

GUI: Analysis interface



GUI: Analysis interface (2)

Features:

- tree represents an analysis:
  - each node is labelled with the result of an analysis step
  - each analysis step corresponds to the execution of a tool
- parameters can be supplied to tools using a graphical interface
- analysis trees abstract from temporary files: treated as cache

Graphical simulator

Features:

- simulate LPEs
- pluggable views

Demo of the parking garage

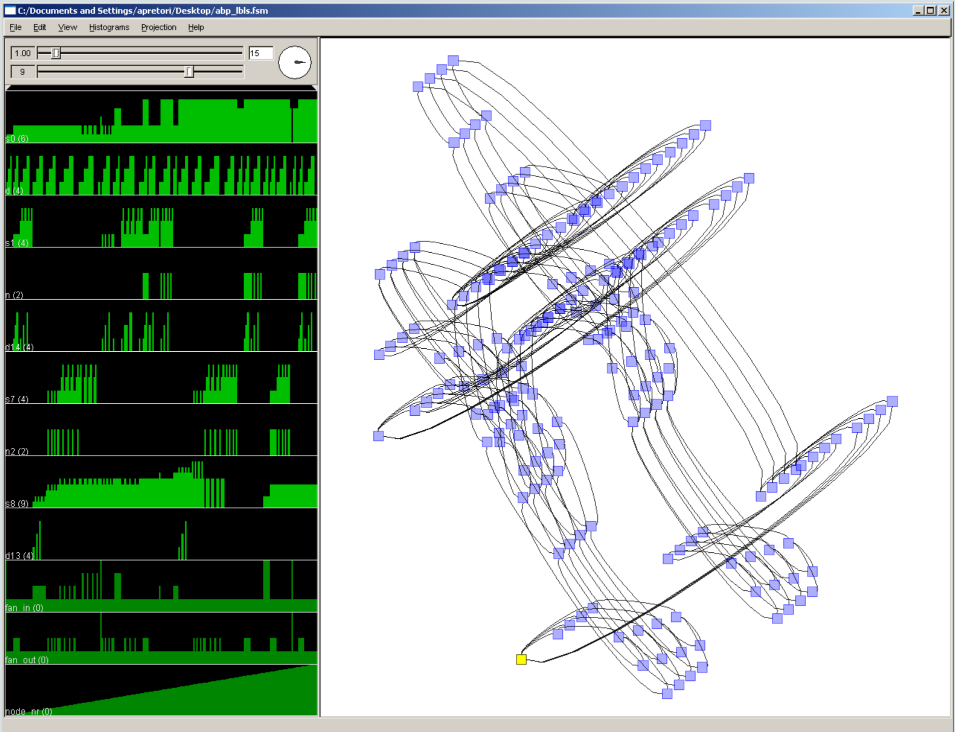
Model checking on LPEs

Parameterised Boolean Equation Systems (PBESs): mixture of BES and HOAS

Check a property P on an LPE E:

1. combine P and E into a PBES
2. convert the PBES to a BES
3. check the BES

Visualisation of large LTSs (Hannes Pretorius)



Tool development status

Finished (mostly):

- parser
- type checker
- implementation of concrete data types
- lineariser
- rewriter (interpreting, compiling, JITty)
- simulator (both textual and graphical)
- instantiator
- 2D LTS visualiser
- classical Petri net to mCRL2 convertor

Tool development status (2)

To be implemented:

- LPE reduction tools
- LPE model checker
- graphical analysis interface
- prover
- μCRL to mCRL2 convertor and vice versa
- coloured Petri net to mCRL2 convertor

Conclusions and future work

mCRL2 is an attempt to make μCRL more applicable in practise.
It is extended such that:

- Petri nets can be facilitated
- the treshold for new users is lowered

Future work:

- formalise the syntax and semantics of mCRL2
- finish the toolset and apply it to a number of real world cases
- find a connection with other toolsets