GenSpect

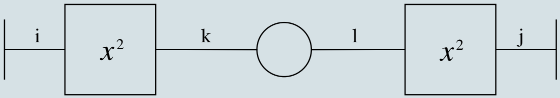
A specification language for Petri Nets and process algebra

Aad Mathijssen

25th May 2005

Motivation

Bring stand-alone developments of specification languages together.
Starting point: find a common base for 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 (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
- component-based approach enforces the need for local communication
- coloured Petri Nets often use a higher-order data language instead of a first-order language

GenSpect language

Distinguish two levels:

- low-level: μCRL-like language (developed by the OAS group)
- high-level: Petri Net-like language focusing on modular design (developed by the IS group)

Requirement: every high-level specification can be translated to a low-level specification

Low-level GenSpect

Process algebra with data, which is basically timed μCRL with the following additions:

- true concurrency (concurrency in Petri Nets)
- local communication (modularity of HL-GenSpect)
- higher-order abstract data types (colours in Petri Nets)

We will call this language mCRL2.

True concurrency

Changes with respect to μCRL:

- sync operator $|$ does not communicate anymore
- a sync of actions is called a *multi-action*, e.g.
  $a,\ a|b,\ b|a,\ a|b|c,\ a|b|a,\ a(t)|b(u)|a(v)$
- added communication operator Γ for the communication of multi-actions, e.g. where $t = u$ and $t \neq v$:
  $\Gamma_{\{a|b \to c\}}(a(t)|b(u)) = c(t),\ \Gamma_{\{a|b \to c\}}(a(t)|b(v)) = a(t)|b(v),$
  $\Gamma_{\{a|b|c \to d\}}(a|b|c|d) = d|d$
- added visibility operator ∇ to restrict behaviour of multi-actions, e.g.
  $\nabla_{\{a,b\}}(a \parallel b) = a \cdot b + b \cdot a,\ \nabla_{\{a|b\}}(a \parallel b) = a|b,$
  $\nabla_{\{a|b\}}(a|b|c) = \delta,\ \nabla_{\{a,b|c\}}(a \parallel b \parallel c) = a \cdot (b|c) + (b|c) \cdot a$

Γ also implements local communication.

mCRL2 process language

Process expressions have the following syntax:

$$\begin{aligned} p ::= {} & a \mid \delta \mid \tau \mid p + p \mid p \cdot p \mid p \parallel p \mid p \mathbin{⌊\!⌊} p \mid p|p \mid X \\ & \mid a(\vec{d}) \mid (d = d) \rightarrow p, p \mid p \cdot d \mid X(\vec{d}) \mid \textstyle\sum_{\overrightarrow{x:s}} p \\ & \mid \nabla_V(p) \mid \Gamma_C(p) \mid \partial_H(p) \mid \tau_I(p) \mid \rho_R(p) \end{aligned}$$

Process equations are formed as follows:

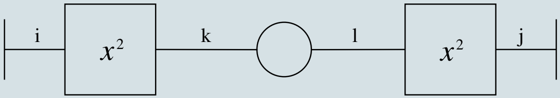
$$pe ::= X = p \mid X(\overrightarrow{x : s}) = p$$

Process specifications:

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

Petri Net translation

Petri Nets can be expressed in mCRL2:



Translation to mCRL2:

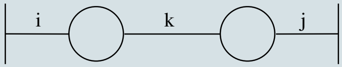
$$
\begin{array}{ll}
Sqr_{i,o} & = \sum_{n:\mathbb{N}} \overline{get_i}(n) | \overline{put_o}(n^2) \cdot Sqr_{i,o} \\
P_{i,o}(b : Bag(\mathbb{N})) & = \sum_{n:\mathbb{N}} \underline{put_i}(n) \cdot P_{i,o}(b \cup \{ n \}) + \\
 & \sum_{n:\mathbb{N}} n \in b \rightarrow \underline{get_o}(n) \cdot P_{i,o}(b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

where

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

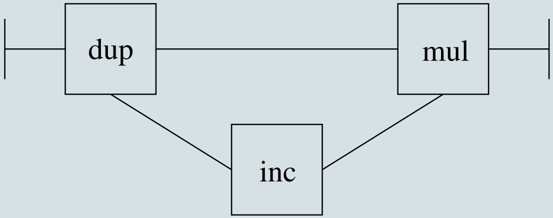
Beyond Petri Nets

Connected places:



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

Connected transitions:



mCRL2 data language

Problems with the current μCRL data language:

- first-order language (coloured Petri Nets)
- lack of concrete data types with a comfortable syntax

Problems with existing data languages:

- 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.

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:

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

Sugaring the data language

For the purpose of user-friendliness, we add sugar:

- $s_0 \times \cdots \times s_n \to s$ is a shorthand for $s_0 \to \cdots \to s_n \to s$, where $\to$ 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:

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

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

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 $\mathbb{Z}$)
  $0, 1, -1, 2, -2, \ldots$
  $d < d, d \leq d, d > d, d \geq d, -d, d + d, d - d, d * d, d\ \mathbf{div}\ d, d\ \mathbf{mod}\ d, \ldots$

Concrete data types (2)

Type constructors:

- structured types (sum types and product types)

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

- 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: Sliding Window Protocol

```
map  n: Pos;

sort D = struct d1 | d2;
     Buf = Nat -> struct data(getdata:D) | empty;
map  emptyBuf: Buf;
     insert: D#Nat#Buf -> Buf;
     remove: Nat#Buf -> Buf;
     release: Nat#Nat#Buf -> Buf;
     nextempty: Nat#Buf -> Nat;
     inWindow: Nat#Nat#Nat -> Bool;
var  i,j,k: Nat; d: D; q: Buf;
eqn  emptyBuf = lambda j:Nat.empty;
     insert(d,i,q) = lambda j:Nat.if(i==j,data(d),q(j));
     remove(i,q) = lambda j:Nat.if(i==j,empty,q(j));
     release(i,j,q) =
        if((i mod 2*n)==(j mod 2*n),
           q,
           release((i+1) mod 2*n,j,remove(i,q)));
     nextempty(i,q) = if(q(i)==empty,i,nextempty((i+1) mod n,q));
     inWindow(i,j,k) = (i<=j && j<k) || (k<i && i<=j) || (j<k && k<i);
```

Example: Sliding Window Protocol (2)

```
act  sA,rA,sD,rD: D;
     sB,rB,cB,sC,rC,cC: D#Nat;
     sE,rE,cE,sF,rF,cF: Nat;
     j;

proc S(l,m:Nat,q:Buf) =
        sum d:D. inWindow(l,m,(l+n) mod 2*n) ->
                rA(d).S(l,(m+1) mod 2*n,insert(d,m,q))+
        sum k:Nat. (q(k)!=empty) -> sB(getdata(q(k)),k).S(l,m,q)+
        sum k:Nat. rF(k).S(k,m,release(l,k,q));

     R(l:Nat,q:Buf) =
        sum d:D,k:Nat. rC(d,k).
                (inWindow(l,k,(l+n) mod 2*n) -> R(l,insert(d,k,q)),R(l,q))+
        (q(l)!=empty) -> sD(getdata(q(l))).R((l+1) mod 2*n,remove(l,q))+
        sE(nextempty(l,q)).R(l,q);

     K = sum d:D,k:Nat. rB(d,k).(j.sC(d,k)+j).K;

     L = sum k:Nat. rE(k).(j.sF(k)+j).L;

init allow({cB,cC,cE,cF,j,rA,sD},
        comm({rB|sB->cB, rC|sC->cC, rE|sE->cE, rF|sF->cF},
          S(0,0,emptyBuf) || K || L || R(0,emptyBuf)));
```

Tool support

Goals:

- provide functionality comparable to that of the μCRL toolset; in particular the concept of linear process equations (LPEs) play a central role
- simplify the process of analysing specifications

Because of all changes and additions, reusing the existing μCRL toolset is almost impossible. Furthermore, there are other changes:

- added time (discrete/continuous, absolute)
- added don't care values

Development status

Finished (mostly):

- parser (Aad)
- type checker (Yaroslav)
- implementation of concrete data types (Aad)
- lineariser (Jan Friso, Muck)
- rewriter (Muck)
- nextstate (Muck) (μCRL: stepper)
- findsolutions (Muck) (μCRL: enumerator)
- simulator (Muck) (both textual and graphical)
- instantiator (Muck)

Development status (2)

To be implemented:

- LPE model checker using the techniques of parameterised boolean equation systems (PBESs) (Jan Friso, Muck)
- LPE reduction tools (?)
- graphical analysis interface (Jan Friso, Aad, Muck)
- prover (Jaco)
- Petri Net to mCRL2 convertor (Yaroslav, Jofra)

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

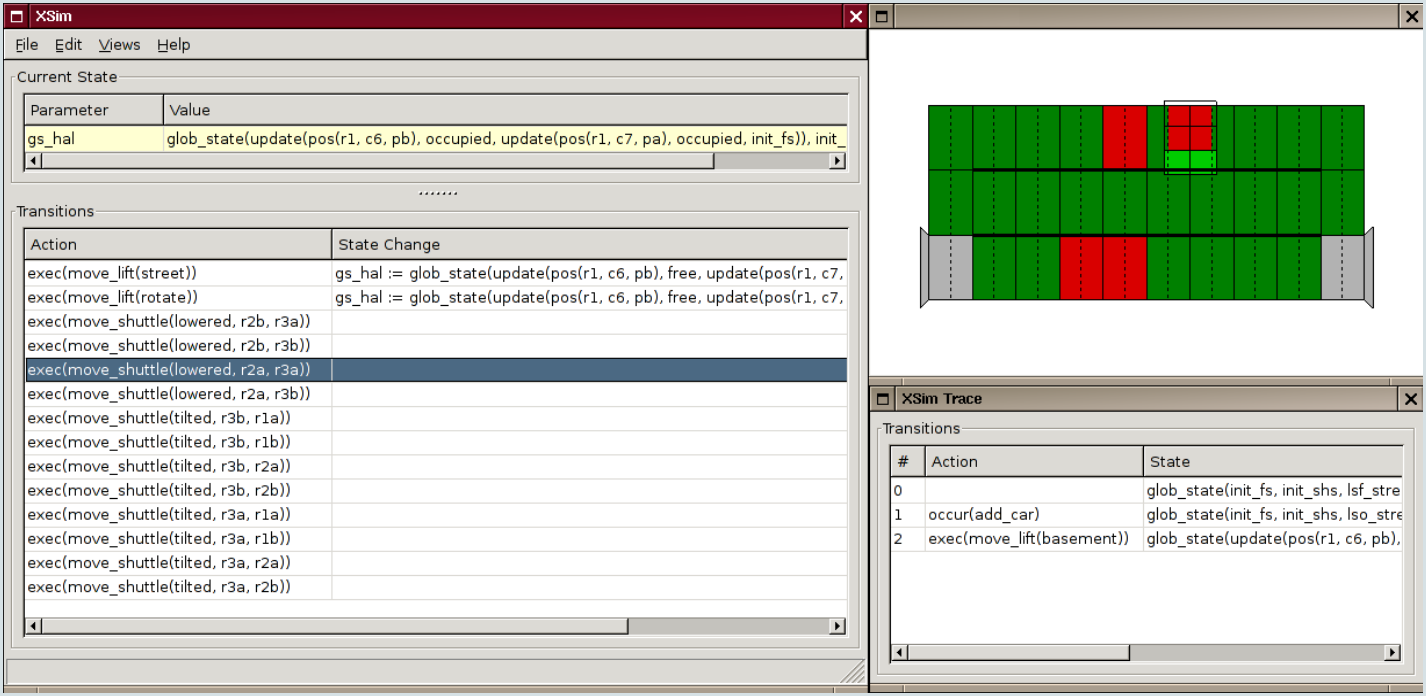
Implementation of concrete data types (2)

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 $(\mathbb{B} \to s) \to s$
- 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 $s \to \mathbb{N}$

Graphical simulator

Features: simulate LPEs, different views

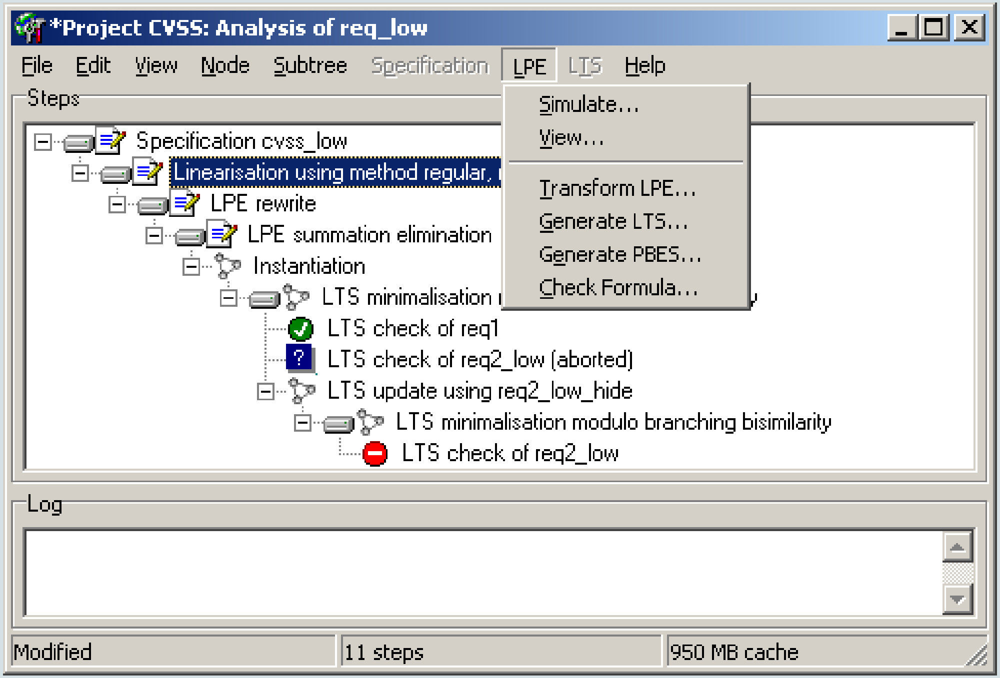


Analysis interface

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

Analysis interface (2)



Conclusions and future work

GenSpect brings together the worlds of Petri Nets and proces algebra.

μCRL 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
- define a translation from HL-GenSpect to LL-GenSpect
- finish mCRL2 toolset and apply it to a number of real world cases
- find a connection between the toolsets of μCRL and mCRL2