Distributed State Space Generation and Minimization

Jaco van de Pol
Bert Lisser, Stefan Blom, Simona Orzan

Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands
SENVA partner

SENVA meeting
November 2005, Grenoble

Overview

1. The μCRL project
2. Verification Approach
3. Distributed State Space Generation
4. Distributed State Space Minimization
5. Implications for GRID

μCRL

μCRL = abstract datatypes + process algebra

- ADT: constructors + maps + equations
- Process algebra: ACP style

Most intriguing construct: Σ - potentially infinite choice.

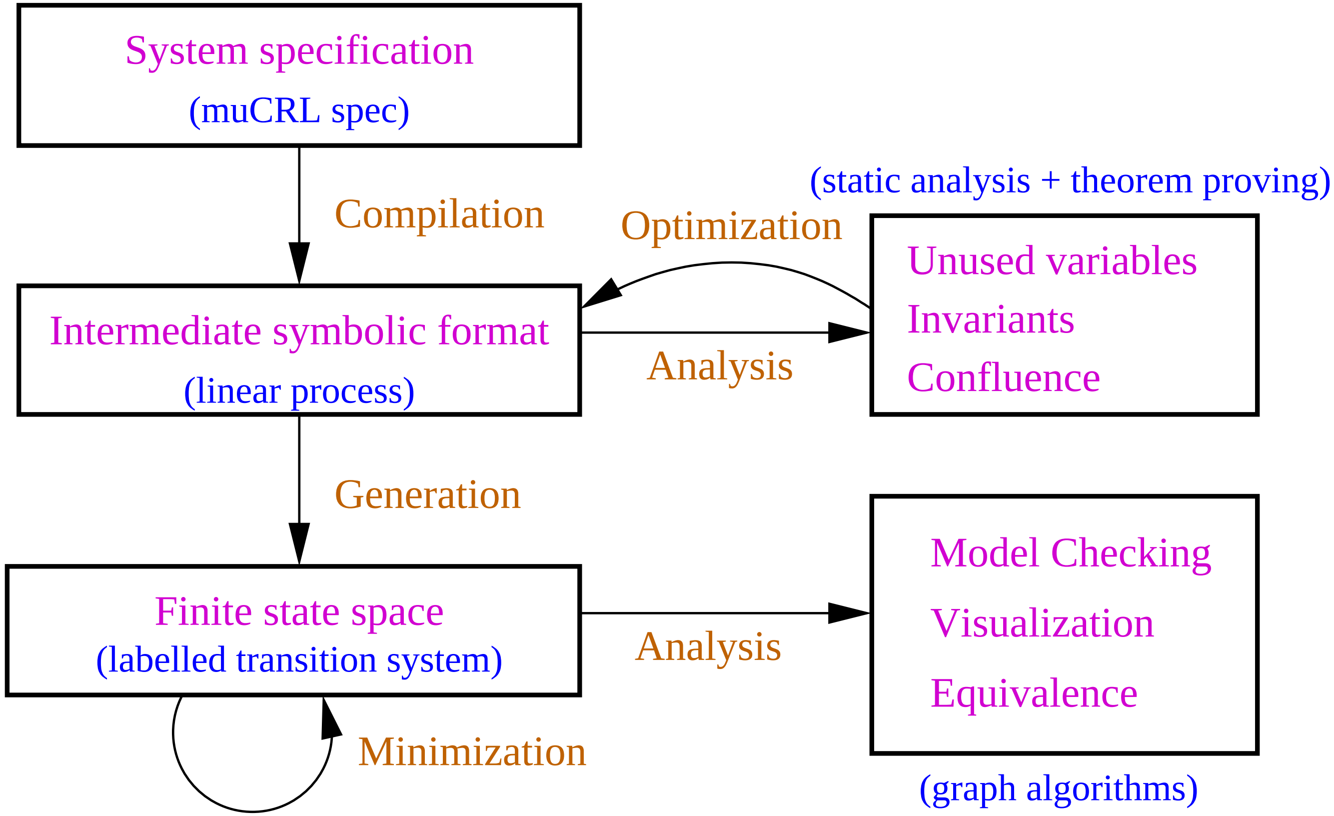
Example from Security Protocols:

$$Alice \,||\, Intruder(K)$$

$$\Big(\sum_m recv(m) \triangleleft protocol(m)\Big) \,||\, \Big(\sum_m send(m) \triangleleft synthesize(m, K)\Big)$$

Techniques: Term rewriting + enumeration (= narrowing?)
(enumerate m such that $protocol(m) \wedge synthesize(m, K)$)

Verification in the μCRL toolset



Fight State Space Explosion!

Linear Process Equations

- First, a μCRL specification is linearized.
  (this step eliminates $||$ and $\cdot$ at the expense of adding data)
- A linear process has the form:

$$
\begin{array}{rcl}
P(\vec{x}) & = & \sum_{\vec{y}} . a_1(\vec{x},\vec{y}) \cdot P(g_1(\vec{x},\vec{y})) \triangleleft b_1(\vec{x},\vec{y}) \\
 & + & \cdots \\
 & + & \sum_{\vec{y}} . a_n(\vec{x},\vec{y}) \cdot P(g_n(\vec{x},\vec{y})) \triangleleft b_n(\vec{x},\vec{y})
\end{array}
$$

- Here $\vec{x}$ is the state vector, containing state variables of all components + program counters. a_i are actions, b_i Boolean guards, g_i next states, and $\vec{y}$ are local choice parameters.
- Advantage: simple structure + relatively succinct
- Symbolic state space reductions are LPE transformations.

Symbolic Optimizations

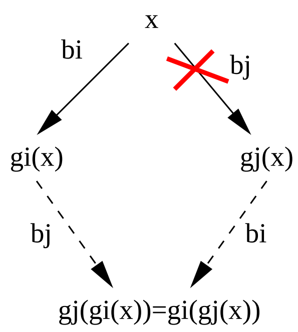
- Constant propagation, Resetting dead variables
- Dead code elimination, tau-confluence reduction

I is an invariant for all summands:

$$\bigwedge_i \forall \vec{x}, \vec{y} : I(\vec{x}) \wedge b_i(\vec{x}, \vec{y}) \Rightarrow I(g_i(\vec{x}, \vec{y}))$$

Summand i commutes with j (for confluence reduction):

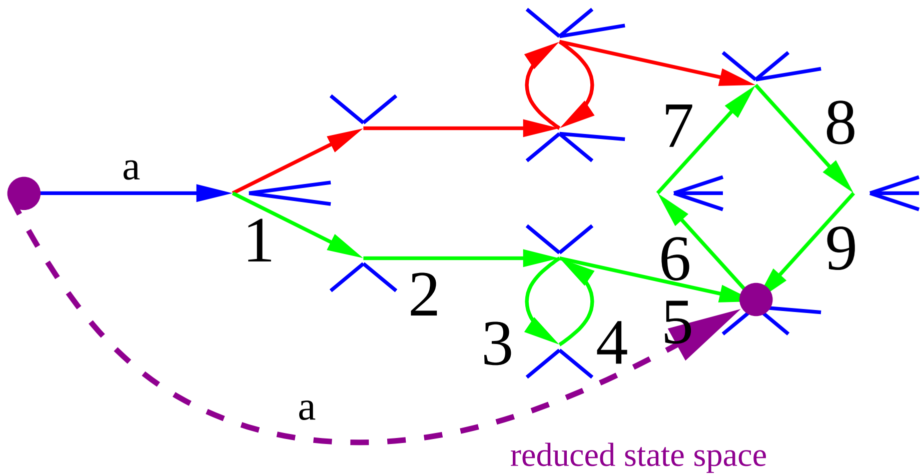
$$\forall \vec{x} : b_i(\vec{x}) \wedge b_j(\vec{x}) \Rightarrow b_j(g_i(\vec{x})) \wedge b_j(g_i(\vec{x})) \wedge g_j(g_i(\vec{x})) = g_i(g_j(\vec{x}))$$



- Invariant generation / checking
- Confluence detection by theorem proving
- Confluence reduction on the fly (avoid loops!)
- $\Rightarrow$ Parallellization of theorem prover ??

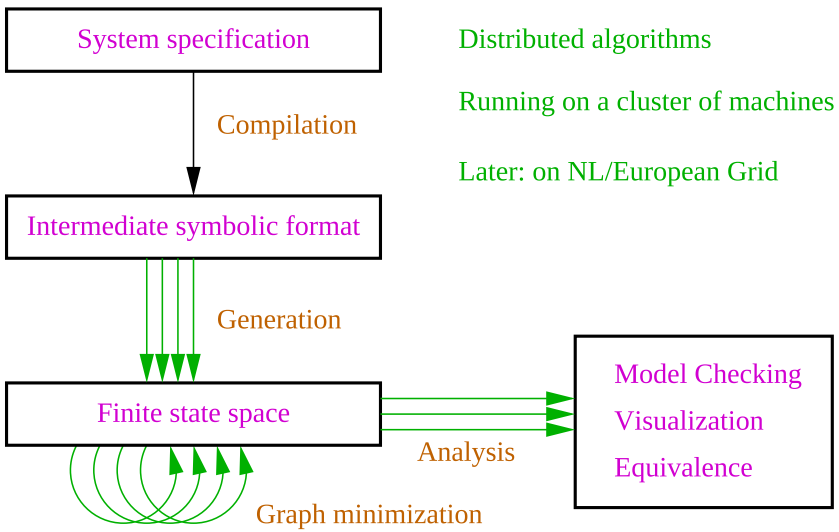
On-the-fly τ-Confluence Reduction

- After detecting some confluent τ summands, we want on the fly:
  - Give priority to confluent τ-steps
  - Compress sequences of confluent τ-steps
- A concrete transition $s \rightarrow_a t$ is transformed to $s \rightarrow_a rep(t)$, where $rep(t)$ is found by Tarjan's SCC algorithm.
- Below: blue are visible steps, while green (visited) and red (not visited) are confluent tau-steps



Explicit state space generation

- When symbolic reduction is exhausted, we start brute force
- Generating state space is time consuming (narrowing!)
- Hence: explicit generation + storage of full state space.
- Various analyses can be performed without regeneration
- Use distributed generation to scale in memory + time.



Distributed State Space Generation

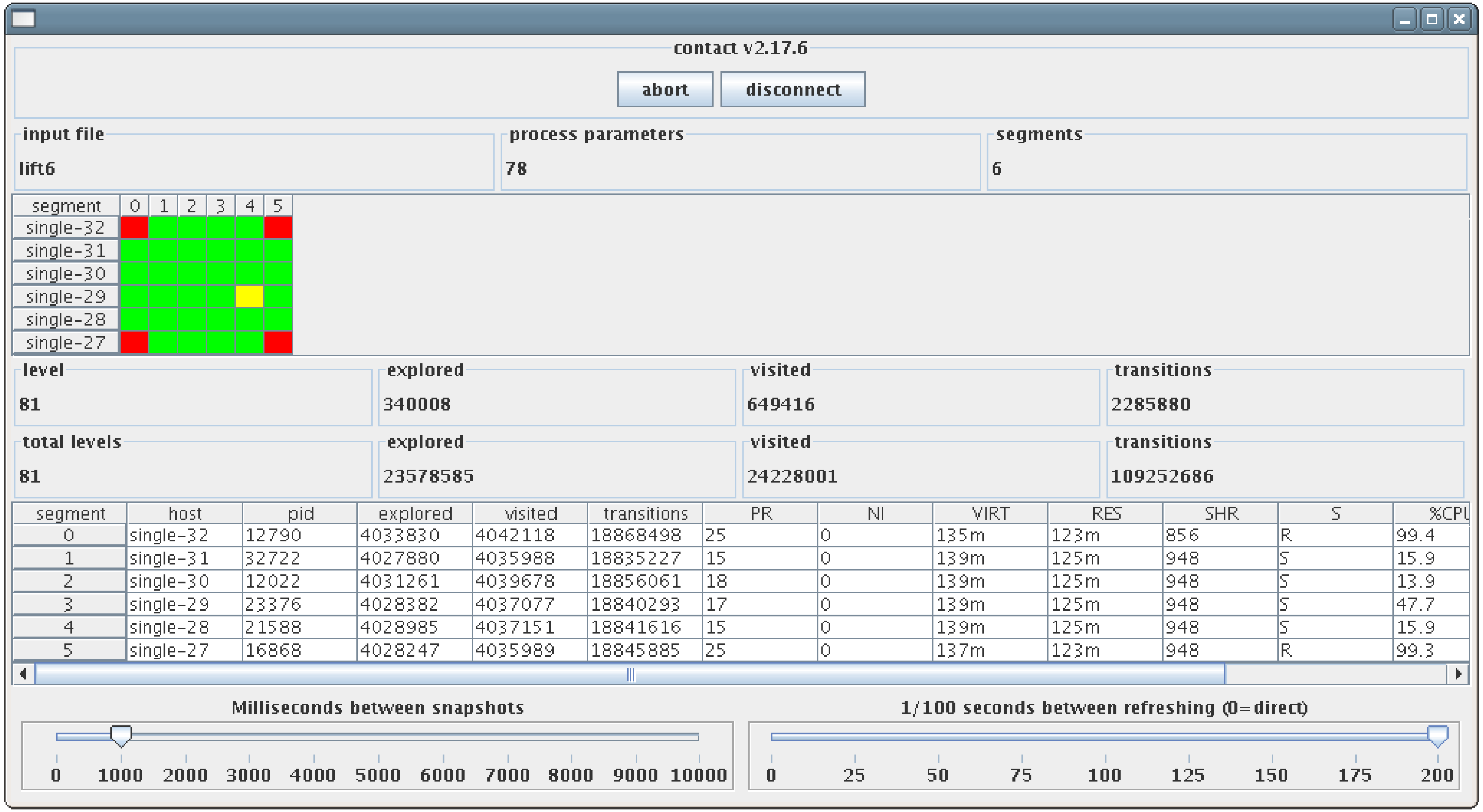
Extra Functionality:

- on-the-fly verification of simple safety properties (deadlock, occurrences of bad actions)
- debug traces for diagnostics
- on-the-fly confluence reduction
- search in time slices to find shortest schedules (i.e.: barrier synchronization on special "tick" actions)

Conclusion:

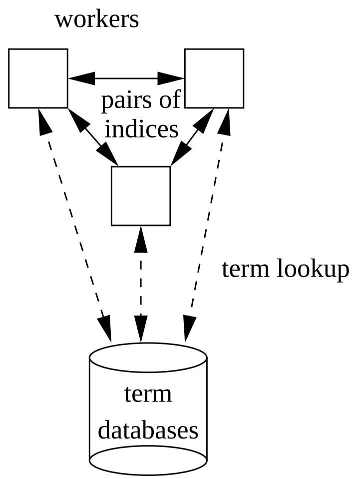
- Scales up in time + memory ($> 10^8$ transitions, 32 nodes)

Distributed State Space Generation



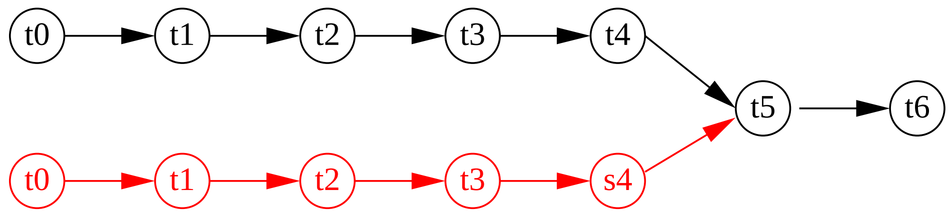
Distributed Generation with On-the-fly τ-Confluence Reduction

- Strict breadth first exploration of the state space.
  (only reason: shortest traces as counter example)
- As usual: states are allocated by a static hash function to nodes in a network, who send each other batches of successor states.
- Actually, we send indices of states to avoid serialization overhead.

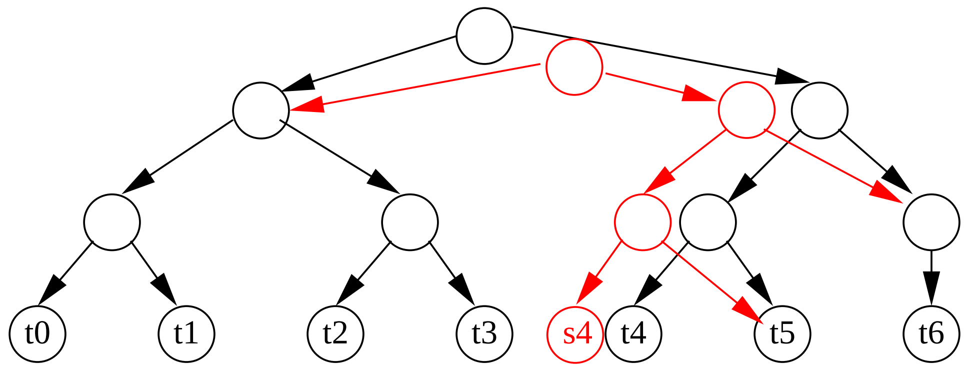


Single node Memory Footprint

- We use the ATerm Library (CWI, SEN 1) to represent the state space as one maximally shared forest.
- Nice trick (JF Groote): arrange state vector as tree instead of list:
- List arrangement: avg. $\frac{1}{2}n$ list nodes duplicated $t4 \mapsto s4$



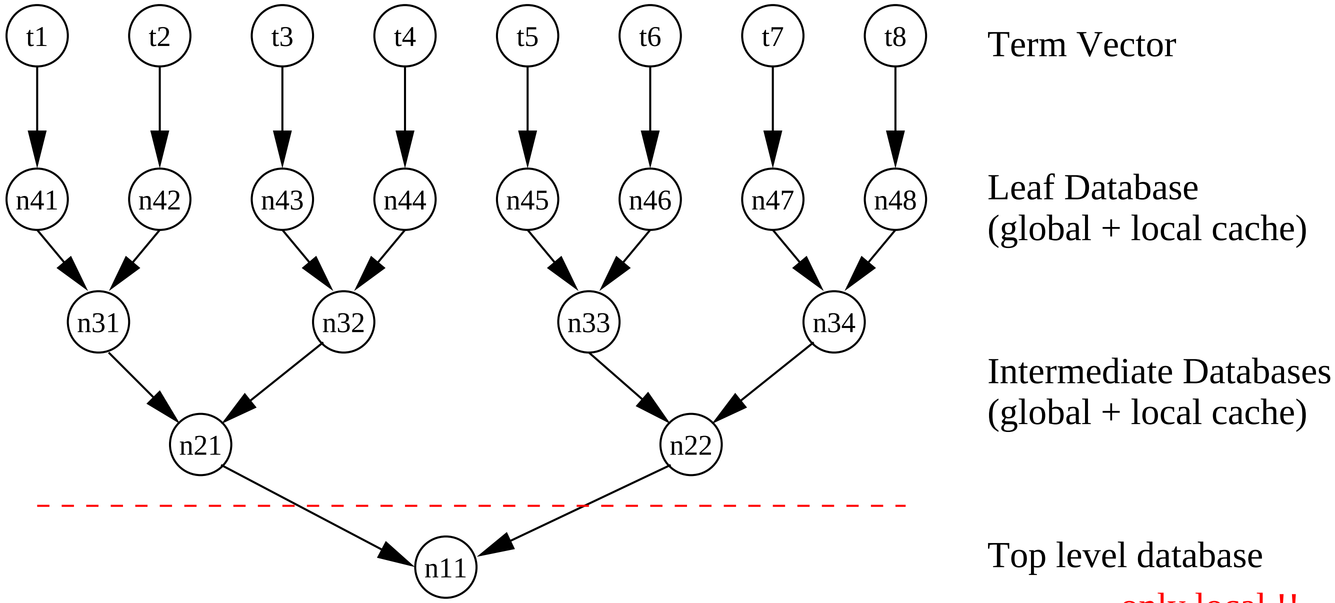
- Tree arrangement: $\log n$ list nodes duplicated.



Distributing shared terms?

- The nodes send each other batches of new states to be explored.
- Problem: ATerms are pointers, which are only locally meaningful.
- Bad solution: serialization/deserialization for communication.
- Our solution:
  - Assign indices to states, only communicate indices
  - Store term indices in a shared global database
  - Exploit another tree-folding trick to avoid bottlenecks
- Goal: a global unique bijection: States $\leftrightarrow \{0, \ldots, n\}$
  (but avoid a database of size n)
- Idea: Use the Cartesian structure of the state vector

Shared global term databases



- With perfect balancing: database n_{kj} at level k has $\sqrt[k]{|S|}$ states
- Balancing can be improved by pairing/projecting the state vector
- Several ad hoc tricks reduce communication overhead.
- Most communication overhead only at the beginning!

State Space Minimization

- We have the following instances:
  - Strong bisimulation reduction (Blom,Orzan)
  - Branching bisimulation reduction (Blom, Orzan)
  - τ-cycle elimination (Orzan, van de Pol)
- Algorithms:
  - Partition refinement, based on "observable signatures".
  - The nodes synchronize in "rounds"
- Conclusion:
  - it works in practice, memory usage is OK
  - decent speedup for large enough examples
  - result after minimization often fits in one machine

Implications for GRID

- intensive inter-node communication
- synchronized levels for BFS (can be relaxed), and for partition refinement (essential?)
- shared data base
- need for persistent data storage

Implications for Interfaces

- data enumeration is important (for open systems)
- extensions for confluence/partial-order reduction needed
- several symbolic optimizations are language specific.