Model checking the probabilistic π-calculus

Gethin Norman
Oxford University
Computing Laboratory

Catuscia Palamidessi
INRIA Futurs Saclay
LIX

Dave Parker
Oxford University
Computing Laboratory

Peng Wu
CNRS
LIX

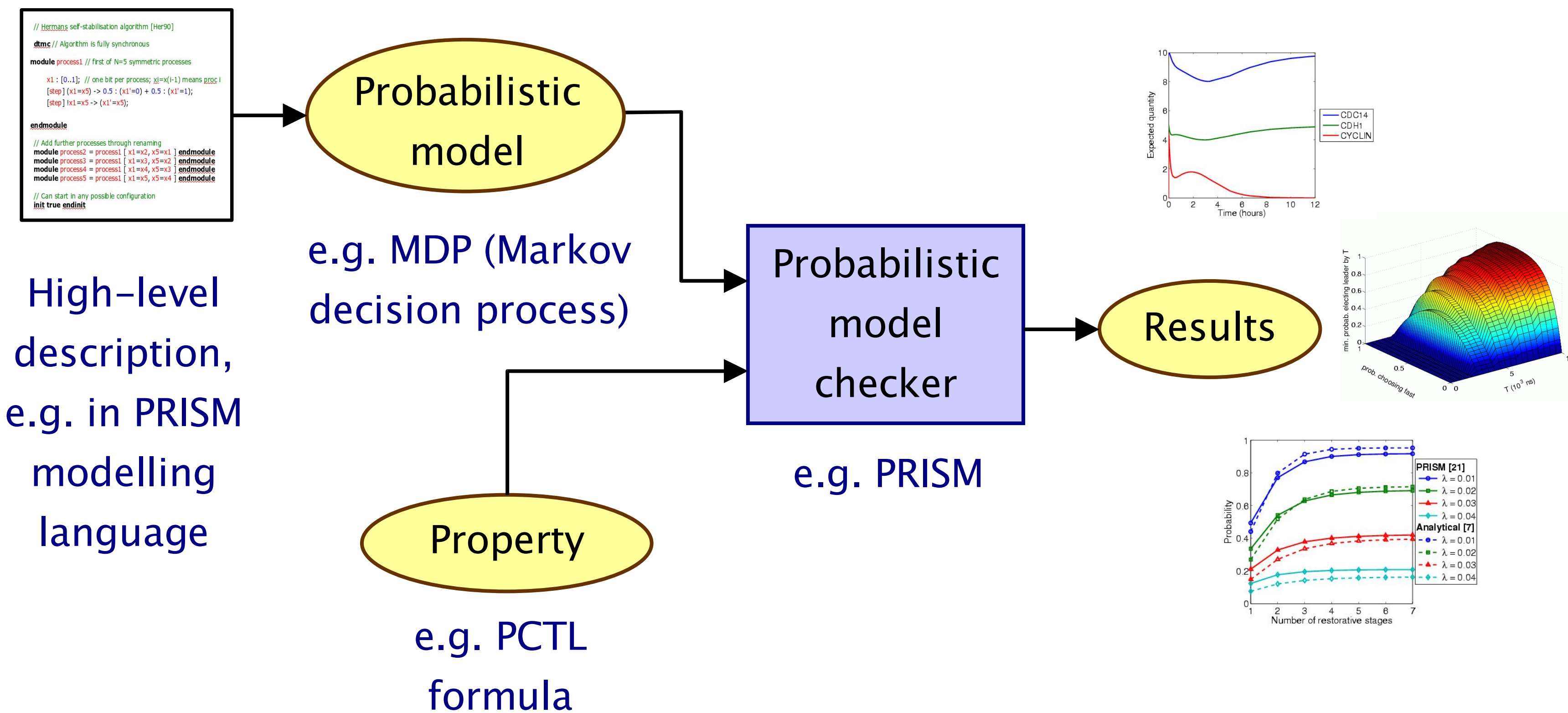
QEST'07

Overview

- Probabilistic model checking
  - Markov decision processes, PCTL, PRISM
- The probabilistic π-calculus
  - syntax, symbolic semantics, example
- π-calculus tool support: MMC
- Adding π-calculus support to PRISM
  - extending MMC with probabilities
  - a compositional approach: translation to PRISM
- Experimental Results
- Conclusions

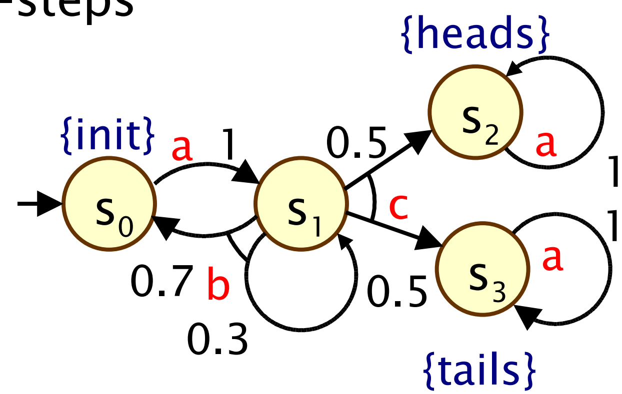
Probabilistic model checking

- Automatic formal verification technique for analysis of systems exhibiting probabilistic behaviour



Markov decision processes (MDPs)

- Model supporting probabilistic and nondeterministic choice
  - discrete state space and discrete time-steps
  - nondeterministic choice between (action-labelled) probability distributions over successor states



- Well suited to modelling of:
  - randomised distributed algorithms, probabilistic communication/security protocols, ...
- Verification using e.g. the logic PCTL
  - $P_{min=?}$ [$F^{\leq t}$ reply_count=k {"init"}{min}]
    "what is the minimum probability, from any initial configuration and under any scheduling, that the sender has received k acknowledgements within t time units?"

PRISM modelling language

- Simple, state-based language for MDPs (and D/CTMCs)
  - based on Reactive Modules [Alur/Henzinger]
- Modules (system components, composed in parallel)
- Variables (finite-valued – integer ranges or booleans)
- Guarded commands (labelled with probabilities/rates)
- Composition of modules: synchronisation (CSP-style) + process-algebraic operators (e.g. action hiding/renaming)

$$\underbrace{[\mathit{send}]}_{\text{action}}\ \underbrace{(s=2)}_{\text{guard}} \mathrel{-\!>} \underbrace{p_{loss}}_{\text{probability}} : \underbrace{(s'=3)\&(lost'=lost+1)}_{\text{update}} + \underbrace{(1-p_{loss})}_{\text{probability}} : \underbrace{(s'=4)}_{\text{update}};$$

The π-calculus

- The π-calculus [Milner et al.]
  - process algebra for concurrency and mobility
  - single datatype, names, for both channels and variables
  - allows dynamic creation of new channel names and communication of channel names between processes
  - ...and therefore dynamic communication topologies
  - applications: e.g. cryptographic protocols, mobile communication protocols, ...
- Probabilistic π-calculus [Herescu/Palamidessi, ...]
  - adds discrete probabilistic choice for modelling of random choice (e.g. coin toss) or unpredictability (e.g. failures)
  - applications: e.g. randomised security protocols, mobile ad-hoc network protocols, ...

Simple probabilistic π−calculus: π_{sp}

[Chatzikokolakis/Palamidessi]

- Processes: P :: =
  - 0 | α.P | P + P | $\Sigma_i\, p_i\, \tau.P_i$ |
    (null) (prefix) (nondet. choice) (internal probabilistic choice)
  - P | P | νx P | [x=y] P | $A(y_1,\ldots,y_n)$
    (parallel) (restriction) (match) (identifier)
- Actions: α ::=
  - in(x,y) | out(x,y) | τ
    (input on x to y) (output of y on x) (internal)
- Example: $Q := \nu a\,(Q_1 \mid Q_2)$
  - $Q_1 := \nu c\, \nu d\, (\, \tfrac{1}{2}\, \tau.\mathrm{out}(a,c).\mathrm{in}(c,v).0 + \tfrac{1}{2}\, \tau.\mathrm{out}(a,d).\mathrm{in}(d,w).0\,)$
  - $Q_2 := \nu b\, (\, \mathrm{in}(a,x).\mathrm{out}(b,x).0 \mid \mathrm{in}(b,y).\mathrm{out}(y,e).0\,)$

Simple probabilistic π-calculus: π_{sp}

- "Simple" refers to restriction to "blind" probabilistic choice
  - "sufficient" modelling power, but simpler semantics/analysis
- Restrictions for model checking
  - finite control (no recursion within parallel composition)
  - input closed (no inputs from environment)
- Semantics are in terms of Markov decision processes
  - or, equivalently, (simple) probabilistic automata [Segala/Lynch]
- We use a symbolic semantics approach
  - often better suited to proof systems, tool support
  - extension of non-probabilistic case [Lin'00,Lin'03]
  - probabilistic symbolic transition graphs (PSTGs)

Symbolic semantics

- A PSTG is a tuple (S, s_{init}, T) where:
  - S is a set of symbolic states (π–calculus processes)
  - $s_{init} \in S$ is the initial state
  - $T \subseteq S \times Cond \times Act \times Dist(S)$ are transitions
- And:
  - Cond is the set of conditions
    - finite conjunctions of matches (name comparisons)
  - Act is the set of actions:
    - τ, in(x,y), out(x,y), b_out(x,y) for names x, y

For a transition:

$$(Q, M, \alpha, \{ p_i : Q_i \}) \in T$$

written:

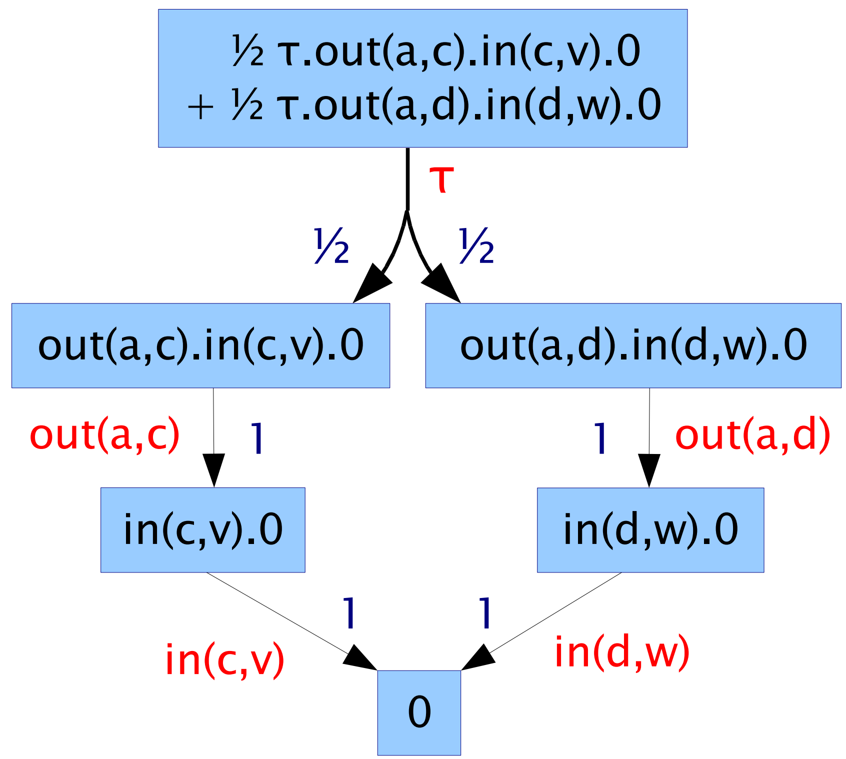
$$Q \xrightarrow{M,\alpha} \{ p_i : Q_i \}$$

"If M is true, Q can perform action α and then with probability p_i evolve as Q_i"

Symbolic semantics

- A PSTG is a tuple (S, s_{init}, T) where:
  - S is a set of symbolic states (π−calculus processes)
  - $s_{init} \in S$ is the initial state
  - $T \subseteq S \times Cond \times Act \times Dist(S)$ are transitions
- And:
  - Cond is the set of conditions
    - finite conjunctions of matches (name comparisons)
  - Act is the set of actions:
    - τ, in(x,y), out(x,y), b_out(x,y) for names x, y

Example:



(empty) conditions omitted

MMC: Mobility Model Checker

- Model checker for (finite control subset of) π–calculus
  - against alternation-free π–μ–calculus
- Efficient implementation based on logic programming (XSB)
  - names in π–calculus are represented as LP variables
    - semantics of names matches variable handling in LP resolution
  - direct LP encoding of π–calculus symbolic semantics
    - efficient (XSB tabled resolution) and provably correct
- Other features of MMC:
  - identifies (some) state equivalences (structural congruence)
  - symmetry reduction: associativity/commutativity of parallel
  - additional support for spi–calculus

Translation – Part 1

- MMC_{sp}: extension of MMC to support π_{sp}
  - add probabilistic version of choice operator
    - direct encoding of semantics, as for other operators
    - modify “trans” rule of MMC to include (textual) probabilities
  - add explicit generation/export of PSTG
  - also identifies free/bound names
- For input-closed process, direct input into PRISM
  - PSTG for input-closed process is an MDP
  - either: encode as a single module in PRISM language
  - or: direct input of transition matrix into PRISM
- Provides translation for any π_{sp} process

Translation – Part 2

- Problems:
  - for large models, enumerating state space in this way inefficient
  - product state-space blow-up (at language level)
  - lack of structure/regularity in model (and hence large MTBDDs)

- Solution: a compositional approach to translation
  - 1. assume process of form: $P := \nu x_1 \ldots \nu x_k (P_1 \mid \ldots \mid P_n)$
    - where each P_i contains no instances of ν operator
    - can use structural congruence to get process in this form
  - 2. generate PSTG for each subprocess P_i (using MMC_{sp})
  - 3. translate set of n PSTGs into n PRISM modules
  - 4. final PRISM model is composition of n modules

Translation to PRISM

- Construction of PRISM module for subprocess P_i:
  - one local variable for state (program counter)
  - one local variable per name bounded by input
  - transitions of the PSTG for P_i translated to PRISM commands
- Map names datatype into PRISM's (basic) type system
  - integer variables, integer constant for each free name
- Model channel communication in PRISM
  - π-calculus: binary synchronisation (CCS), name passing
  - PRISM: multi-way synchronisation (CSP), no value passing
  - our translation: encode all information in action names

Example

- $Q := \nu a\,(Q_1 \mid Q_2)$
  - $Q_1 := \nu c\, \nu d\ (\,½\ \tau.out(a,c).in(c,v).0 + ½\ \tau.out(a,d).in(d,w).0\,)$
  - $Q_2 := \nu b\ (\,in(a,x).out(b,x).0 \mid in(b,y).out(y,e).0\,)$

- Rewrite process Q as structurally congruent process P

- $P := \nu a\, \nu b\, \nu c\, \nu d\ (P_1 \mid P_2 \mid P_3)$
  - $P_1 := ½\ \tau.out(a,c).in(c,v).0 + ½\ \tau.out(a,d).in(d,w).0$
  - $P_2 := in(a,x).out(b,x).0$
  - $P_3 := in(b,y).out(y,e).0$

Example – PRISM model structure

$P := \nu a\ \nu b\ \nu c\ \nu d\ (P_1 \mid P_2 \mid P_3)$

$P_1 := ½\ \tau.out(a,c).in(c,v).0$
$\quad + ½\ \tau.out(a,d).in(d,w).0$

$P_2 := in(a,x).out(b,x).0$

$P_3 := in(b,y).out(y,e).0$

Free names in P_1, P_2, P_3:
a, b, c, d, e

Input-bound names:
v, w (P_1), x (P_2), y (P_3)

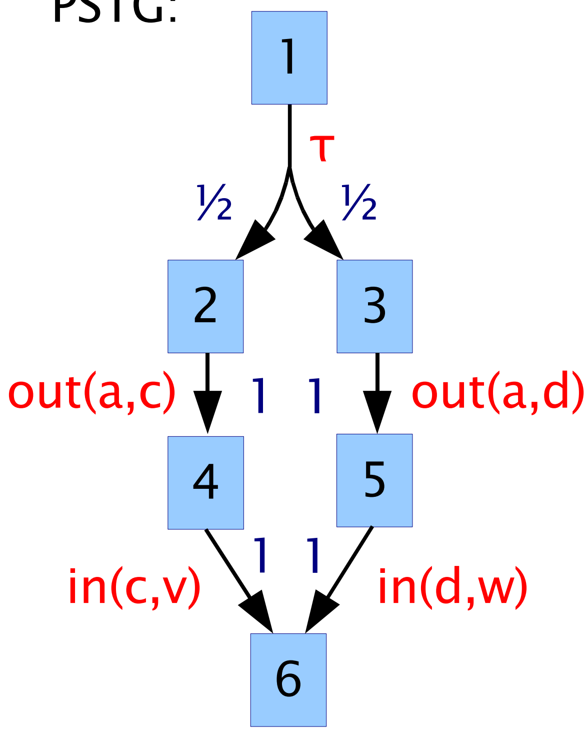
```
const int a = 1; const int b = 2;
const int c = 3; const int d = 4;
const int e = 5;
module P1
    s1 : [1..6] init 1;
    v : [0..5] init 0;
    w : [0..5] init 0;
    ...
endmodule
module P2
    s2 : [1..3] init 1
    x : [0..5] init 0;
    ...
endmodule
module P3
    s3 : [1..2] init 1
    y : [0..5] init 0;
    ...
endmodule
```

Example – A PRISM module

P_1 :=
½ τ.out(a,c).in(c,v).0 +
½ τ.out(a,d).in(d,w).0

Each PSTG transition is mapped to one or more PRISM commands

PSTG:



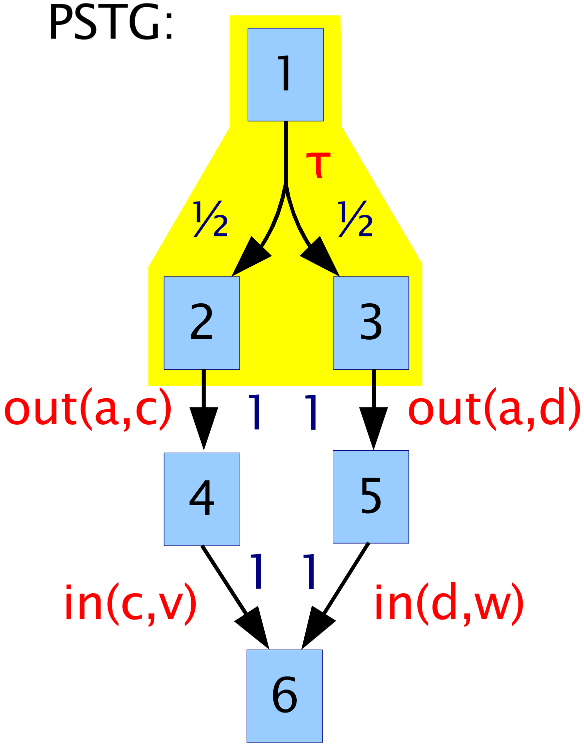

```
module P1
    s1 : [1..6] init 1;
    v : [0..5] init 0;
    w : [0..5] init 0;
    [] (s1 = 1) -> 0.5 : (s1' = 2) + 0.5 : (s1' = 3);
    [a_P1_P2_c] (s1 = 2) -> (s1' = 4);
    [a_P1_P2_d] (s1 = 3) -> (s1' = 5);
    [c_P3_P1_e] (s1 = 4) -> (s1' = 6) & (v' = e);
    [d_P3_P1_e] (s1 = 5) -> (s1' = 6) & (w' = e);
endmodule
```

Example – A PRISM module

P_1 :=
½ τ.out(a,c).in(c,v).0 +
½ τ.out(a,d).in(d,w).0

Each PSTG transition is mapped to one or more PRISM commands

PSTG:



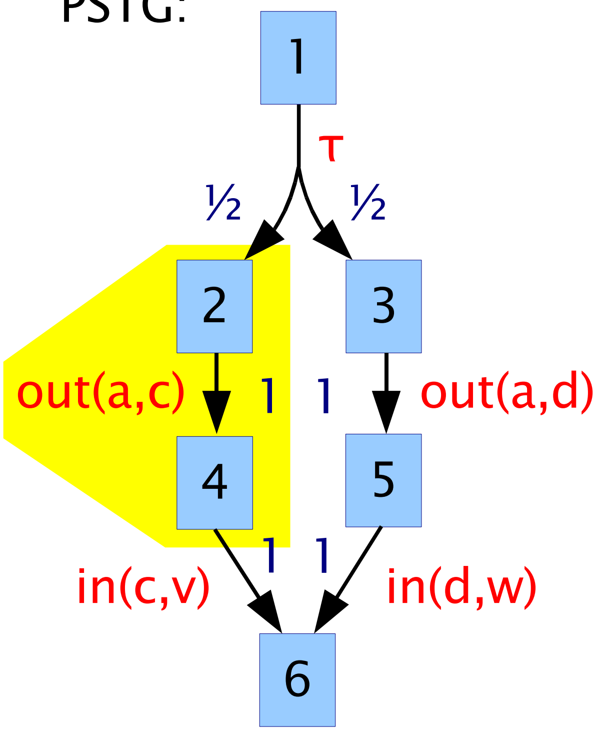

```
module P1
    s1 : [1..6] init 1;
    v : [0..5] init 0;
    w : [0..5] init 0;
    [] (s1 = 1) -> 0.5 : (s1' = 2) + 0.5 : (s1' = 3);
    [a_P1_P2_c] (s1 = 2) -> (s1' = 4);
    [a_P1_P2_d] (s1 = 3) -> (s1' = 5);
    [c_P3_P1_e] (s1 = 4) -> (s1' = 6) & (v' = e);
    [d_P3_P1_e] (s1 = 5) -> (s1' = 6) & (w' = e);
endmodule
```

Example – A PRISM module

P_1 :=
½ τ.out(a,c).in(c,v).0 +
½ τ.out(a,d).in(d,w).0

Each PSTG transition is mapped to one or more PRISM commands

PSTG:




```
module P1
    s1 : [1..6] init 1;
    v : [0..5] init 0;
    w : [0..5] init 0;
    [] (s1 = 1) -> 0.5 : (s1' = 2) + 0.5 : (s1' = 3);
    [a_P1_P2_c] (s1 = 2) -> (s1' = 4);
    [a_P1_P2_d] (s1 = 3) -> (s1' = 5);
    [c_P3_P1_e] (s1 = 4) -> (s1' = 6) & (v' = e);
    [d_P3_P1_e] (s1 = 5) -> (s1' = 6) & (w' = e);
endmodule
```

Example – Module communication

$P_1 :=$
$\frac{1}{2}\ \tau.out(a,c).in(c,v).0\ +$
$\frac{1}{2}\ \tau.out(a,d).in(d,w).0$

```
module P1
    s1 : [1..6] init 1;
    v : [0..5] init 0;
    w : [0..5] init 0;
    [] (s1 = 1) -> 0.5 : (s1' = 2) + 0.5 : (s1' = 3);
    [a_P1_P2_c] (s1 = 2) -> (s1' = 4);
    [a_P1_P2_d] (s1 = 3) -> (s1' = 5);
    [c_P3_P1_e] (s1 = 4) -> (s1' = 6) & (v' = e);
    [d_P3_P1_e] (s1 = 5) -> (s1' = 6) & (w' = e);
endmodule
```

$P_2 := in(a,x).out(b,x).0$

```
module P2
    s2 : [1..3] init 1
    x : [0..5] init 0;
    [a_P1_P2_c] (s2 = 1) -> (s2' = 2) & (x' = c);
    [a_P1_P2_d] (s2 = 1) -> (s2' = 2) & (x' = d);
    [b_P2_P3_x] (s2 = 2) -> (s2' = 3);
endmodule
```

Translation optimisation

- Basic form of translation makes no assumption about which processes can send which names to each other
- For example:
  - action out(x,y) in process P_i for bound x and free y
  - results in a_Pi_Pj_y-labelled command for each $j=1,\ldots,n$ $(j\neq i)$ and each free name a
- In practice, we optimise our translation
  - by computing (an over-approximation of) which processes can send which names to each other
  - with a (finite) iterative analysis of possible values of each input-bound name (and hence each outgoing channel/name)

Property translation

- Currently, we restrict analysis of π_{sp} processes to:
  - (min/max) probabilistic reachability of availability of actions
  - e.g. “minimum probability of getting to state where one of the n subprocesses has reached an error state”
  - easily identified during construction of PSTGs
  - check reachability using PRISM's P=? [F ...] operator

- Possible extensions
  - add test/watchdog processes to system for checking more complex properties
  - expected cost/reward properties

Results

- Implementation: MMC_{sp} + Java translator + PRISM
- 3 case studies from literature:
  - dining cryptographers protocol, partial secrets exchange algorithm, mobile communication network (MCN)
- Largest MDP = 10^9 states = 40 seconds total construction
  - full results in paper
- Analysis of results
  - translation is fast and scalable
  - MCN case study, although small, provides best test of approach
  - efficiency of symbolic (MTBDD) representation from auto-generated PRISM code needs improvement in some cases

Conclusions

- First automated verification of probabilistic π-calculus
  - combination of existing tools: MMC and PRISM
  - encouraging experimental results
- Future work
  - MTBDD efficiency improvements
  - polyadic variants of π-calculus, e.g. out(x,(a,b))
  - automatic translation of (PCTL) properties
  - further properties, e.g. spatial logics, watchdog processes
  - more complex (and bigger) case studies
  - stochastic π-calculus, biological case studies

Full results

Model	N	States	Transitions	MTBDD nodes	Construction time (sec.)			Model checking time (sec.)
					PSTGs	PRISM	MDP	
DCP	5	160,543	592,397	58,641	10.9	0.81	0.77	2.49
	6	1,475,401	6,520,558	100,290	13.1	0.91	1.43	7.82
	7	13,221,889	68,121,834	154,500	15.2	1.17	2.62	21.3
	8	116,192,457	683,937,352	221,170	18.1	1.21	4.72	55.2
	9	1,005,495,499	6,657,256,911	463,425	19.1	1.37	19.3	732.9
PSE	3	9,321	32,052	37,008	4.86	0.75	1.60	1.89
	4	89,025	419,172	103,779	6.60	0.91	3.95	4.47
	5	837,361	5,028,700	173,644	8.12	1.20	8.47	11.5
PSE_3	3	9,328	32,059	37,251	5.29	0.75	2.38	2.16
	4	89,040	419,187	104,267	6.69	0.96	4.19	13.8
	5	837,392	5,028,731	175,212	7.82	1.13	7.58	52.4
MCN	2	609	950	58,430	4.33	2.49	4.8	1.17
	3	3,611	5,811	216,477	5.89	3.11	22.4	5.24

Structural congruences

- For example
  - $P_1 \mid \nu x\, P_2 \equiv \nu x\, (\, P_1 \mid P_2\,)$
  - if x does not occur in P_1