Advanced Tutorial on Bounded Model Checking (BMC)

ACSD'06 - ATPN'06

26th of June 2006

Keijo Heljanko and Tommi Junttila
Keijo.Heljanko@tkk.fi, Tommi.Junttila@tkk.fi

Organisers

- D.Sc. (Tech.), Academy Research Fellow Keijo Heljanko
  - Email: Keijo.Heljanko@tkk.fi
  - Homepage: http://www.tcs.tkk.fi/~kepa/
- D.Sc. (Tech.) Tommi Junttila
  - Email: Tommi.Junttila@tkk.fi
  - Homepage: http://www.tcs.tkk.fi/~tjunttil/
- Our affiliation:
  Laboratory for Theoretical Computer Science,
  Helsinki University of Technology (TKK)

Thanks

Thanks to co-authors on papers related to bounded model checking (in alphabetic order):

- Armin Biere, Johannes Kepler University of Linz
- Toni Jussila, Johannes Kepler University of Linz
- Timo Latvala, University of Illinois at Urbana-Champaign
- Ilkka Niemelä, Helsinki University of Technology (TKK)
- Jussi Rintanen, National ICT Australia Limited (NICTA)
- Viktor Schuppan, ETH Zürich

Tutorial Homepage

- All the material of the Tutorial is available as PDF files from the tutorial homepage:
  http://www.tcs.tkk.fi/~kepa/bmc-tutorial.html
- The PDF files also contain lots of hyperlinks to referenced papers, tools, etc.

Software failures

Software is used widely in many applications where a bug in the system can cause large damage:

- Safety critical systems: airplane control systems, medical care, train signalling systems, air traffic control, etc.
- Economically critical systems: e-commerce systems, Internet, microprocessors, etc.

Price of Software Defects

Two very expensive software bugs:

- Intel Pentium FDIV bug (1994, approximately $500 million).
- Ariane 5 floating point overflow (1996, approximately $500 million).

Pentium FDIV - Software bug in HW

4195835 - ((4195835 / 3145727) * 3145727) = 256

The floating point division algorithm uses an array of constants with 1066 elements. However, only 1061 elements of the array were correctly initialised.

Ariane 5

Exploded 37 seconds after takeoff - the reason was an overflow in a conversion of a 64 bit floating point number into a 16 bit integer.

Finding Bugs in Concurrent Systems

The principal methods for the validation of complex parallel and distributed systems are:

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (mathematical (manual) proof of correctness, in practice done with computer aided proof assistants/theorem provers)
- Model Checking ($\approx$ exhaustive testing of a model of the system)

Why is Testing Hard?

Testing should always be done! However, testing parallel and distributed systems is not always cost effective:

- Testing concurrency related problems is often done only when rest of the system is in place
  $\Rightarrow$ fixing bugs late can be very costly.
- It is labour intensive to write good tests.
- It is hard if not impossible to reproduce bugs due to concurrency encountered in testing.
  - Did the bug-fix work?
- Testing can only prove the existence of bugs, not their non-existence.

Simulation

The main method for the validation of hardware designs:

- When designing new microprocessors, no physical silicon implementation exists until very late in the project.
- Example: Intel Pentium 4 simulation capacity (Roope Kaivola, talk at CAV05):
  - 8000 CPUs
  - Full chip simulation speed 8 Hz (final silicon > 2 GHz).
  - Amount of real time simulated before tape-out: around 2 minutes.

Deductive Verification

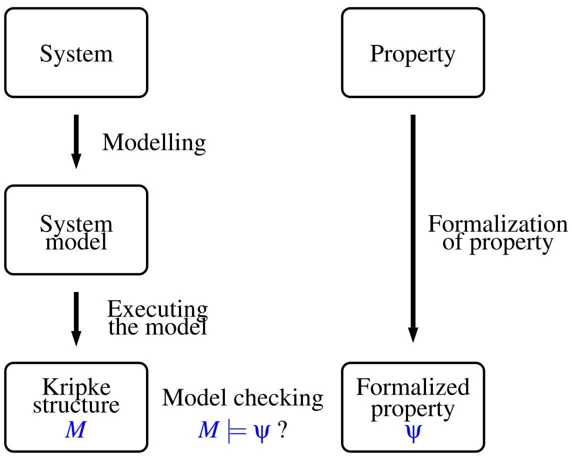
- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants/theorem provers used to keep you honest and to prove sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical systems.
  - HW examples: Pentium 4 FPU, Pentium 4 register rename logic (Roope Kaivola: 2 man years, 2 'time bomb' silicon bugs found - thankfully masked by surrounding logic)

Model Checking

In model checking every execution of the model of the system is simulated obtaining a Kripke structure M describing all its behaviours. M is then checked against a property ψ:

- Yes: The system functions according to the specified property (denoted $M \models \psi$).
  The symbol $\models$ is pronounced “models”,
  hence the term model checking.
- No: The system is incorrect (denoted $M \not\models \psi$), a counterexample is returned: an execution of the system which does not satisfy the property.

Models and Properties



Benefits of Model Checking

- In principle automated: Given a system model and a property, the model checking algorithm is fully automatic.
- Counterexamples are valuable for debugging.
- Already the process of modelling catches a large percentage of the bugs: good for rapid prototyping of concurrency related features.

Drawbacks of Model Checking

- State explosion problem: Capacity limits of model checkers can be exceeded.
- Manual modelling often needed:
  - Model checker used might not support all features of the final implementation language.
  - Abstraction used to overcome capacity problems.

Model Checking in the Industry

- Microprocessor design: Several major microprocessor manufacturers use model checking methods as a part of their design process.
- Design of Data-communications Protocol Software: Model checkers have been used as rapid prototyping systems for new data-communications protocols under standardisation.
- Mission Critical Software: NASA space program is model checking code used by the space program.
- Operating Systems: Microsoft is using model checking to verify the correct use of locking primitives in Windows device drivers.

Modelling Languages

As a language describing system models we can for example use:

- Petri nets,
- labelled transition systems (LTSs) and process algebras,
- Java programs,
- UML (unified modelling language) state machines,
- Promela language (input language of the Spin model checker), and
- VHDL,Verilog, or SMV languages (mostly for HW design).

Some Model Checking Approaches

- Explicit State Model Checking: Tools include Spin, Murφ Java Pathfinder Maria, PROD, CPN Tools, CADP, etc.
- BDD based Symbolic Model Checking: Tools include NuSMV 2, VIS, Cadence SMV, etc.
- Bounded Model Checking: Tools include BMC, CMBC, NuSMV 2, VIS, Cadence SMV, etc.

Bounded Model Checking

- Originally presented in the paper: Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: Symbolic Model Checking without BDDs. TACAS 1999: 193-207, LNCS 1579.
- A closely related approach had already been used earlier to solve artificial intelligence planning problems in: Henry A. Kautz, Bart Selman: Planning as Satisfiability.Proceedings of the 10th European conference on Artificial intelligence (ECAI'92): 359-363, 1992, Kluwer.

Basics of Bounded Model Checking

- The basic idea is the following: Encode all the executions of the system M of length k into a propositional formula $|[M]|^k$.
- Conjunct this formula with a formula $|[\neg\psi]|^k$ which is satisfiable for all executions the system of length k which violate the property ψ.
- If the formula $|[M]|^k \wedge |[\neg\psi]|^k$ is satisfiable, a counterexample has been found.
- If the formula $|[M]|^k \wedge |[\neg\psi]|^k$ is unsatisfiable, no counterexample of length k exists.

SAT

- The propositional satisfiability problem (SAT) is one of the main instances of NP-complete problems.
- Thus no polynomial algorithms for SAT are known.
- However, there are highly efficient SAT solvers available such as zChaff and MiniSAT which are able solve many bounded model checking problems efficiently.

SAT References

- zChaff: Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik: Chaff: Engineering an Efficient SAT Solver. DAC 2001: 530-535, ACM.
- MiniSAT: Niklas Eén, Niklas Sörensson: An Extensible SAT-solver. SAT 2003: 502-518, LNCS 2919.
- SATLive! - Links to SAT related events, tools, position announcements, etc.
- SAT race 2006 - In 2006 a “light weight” variant the SAT solver competition on industrial benchmarks is arranged.

Basic Setup

- For simplicity first consider the following setup:
  - As system models we consider systems whose state vector s consist of n Boolean state variables $\langle s[0], s[1], \ldots, s[n-1]\rangle$.
  - We take $k+1$ copies of the system state vector denoted by $s_0, s_1, \ldots, s_k$.
  - Let $I(s)$ be the initial state predicate of the system, and $T(s, s')$ be the transition relation both expressed as propositional formulas.

A Simplifying Assumption

- For simplicity we assume $T(s,s')$ to be be total for now, i.e., every reachable state s should have a successor s' such that $T(s,s')$ holds.
- This assumption can and will be dropped later in this tutorial.

Unrolling the Transition Relation

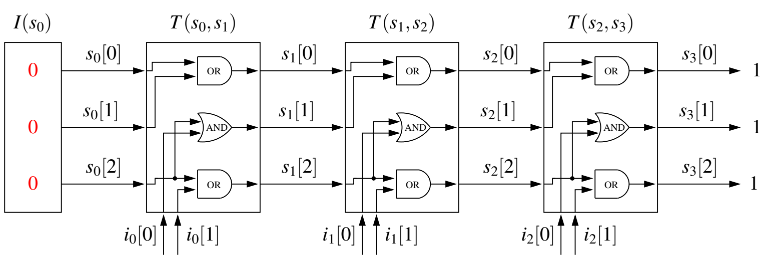
- Now the executions of the system of length k are captured by the formula:

$$|[M]|^k = I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i)$$

- For $k = 3$ this becomes:

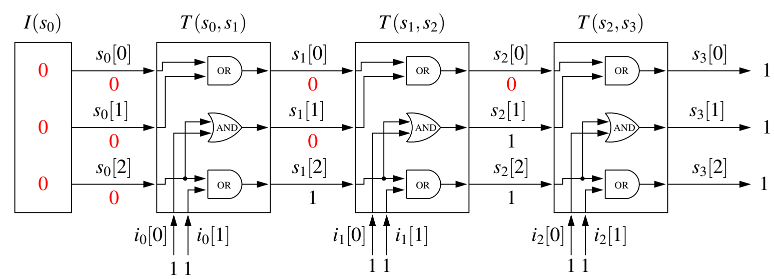
$$|[M]|^3 = I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2) \wedge T(s_2,s_3)$$

Circuit BMC Unrolling



What do the input vectors i_0, i_1, and i_2 need to be to reach the state $s_3 = \langle 1,1,1 \rangle$?

Circuit BMC Unrolling Solution



The input vectors $i_0 = \langle 1,1 \rangle$, $i_1 = \langle 1,1 \rangle$, and $i_2 = \langle 1,1 \rangle$ will reach the final state $s_3 = \langle 1,1,1 \rangle$.

Expressing Invariants

- Suppose the property ψ we want to model check is that an invariant property $P(s)$ holds for every reachable state of the system M.
- Now we get that:

$$|[\neg\psi]|^k = \bigvee_{i=0}^{k} \neg P(s_i)$$

- Thus for $k = 3$ this becomes:

$$|[\neg\psi]|^3 = \neg P(s_0) \vee \neg P(s_1) \vee \neg P(s_2) \vee \neg P(s_3)$$

Final formula

- Thus the final formula $|[M]|^k \wedge |[\neg\psi]|^k$ for $k = 3$ becomes:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3) \wedge$$
$$(\neg P(s_0) \vee \neg P(s_1) \vee \neg P(s_2) \vee \neg P(s_3))$$

- If the formula is satisfiable, then an execution of the system of length 3 exists which violates the invariant property $P(s)$ in some state during the execution.

Reachability Diameter

- If the formula is unsatisfiable, we have proved that there is no execution of length at most 3 that violates the invariant.
- Clearly for every finite state system there is some bound d called the reachability diameter such that from the initial state every reachable state is reachable with an execution of at most length d.
- By taking $d = 2^n$, where n is the number of state bits, we could guarantee completeness.
- Unfortunately computing better approximations of d are computationally hard in the general case.

Unsatisfiable - Increase the bound

- Unfortunately the approach of taking $d = 2^n$ is not viable for anything but trivially small systems.
- Usually d is only increased by a small amount, say 1, and the procedure is repeated from the beginning until some resource limit (running time, memory, etc.) is hit.
- We will show a more refined approach to obtaining completeness later.

BMC: Pros and Cons

- Boolean formulas can be more compact than BDDs
- Leverages efficient SAT-solver technology
- Minimal length counterexamples (often, not always)
- Basic method is incomplete (we'll show some approaches to obtain completeness later in the tutorial)
- Not always better than BDD-based methods or explicit state model checking

Alternative Transition Relations

- When checking for reachability properties such as the violation of invariants, we can often replace the transition relation $T(s,s')$ with an alternative transition relation definition $T'(s,s')$ provided that:
  - Every state that is reachable from the initial state s_0 using $T(s,s')$ must be reachable from s_0 using $T'(s,s')$.
  - There should not be any new states reachable from s_0 using $T'(s,s')$ which are not reachable from s_0 using $T(s,s')$.

Encoding the Transition Relation

- There are now in fact many different ways to pick and encode an alternative transition relation $T'(s,s')$ if we consider asynchronous systems containing concurrency.
- A wish-list of mutually conflicting requirements for $T'(s,s')$ and its encoding:
  - Compact, hopefully linear in the size of the model.
  - Covers as many reachable states as possible for each bound k without losing soundness or completeness.
  - Efficiently solvable by the SAT solver.

Transition Relation Encoding

- Note that in the list of requirements we don't explicitly list that the number of state variables n should be minimised.
- This is often one of the main things to optimise with a BDD based symbolic model checker.
- Having too compact an encoding of the state vector can lead to losses in the SAT solver efficiency!
- More research is needed on how to more efficiently encode transition relations for different classes of systems. There are dramatic performance differences, at least for asynchronous systems.

Asynchronous Systems Case

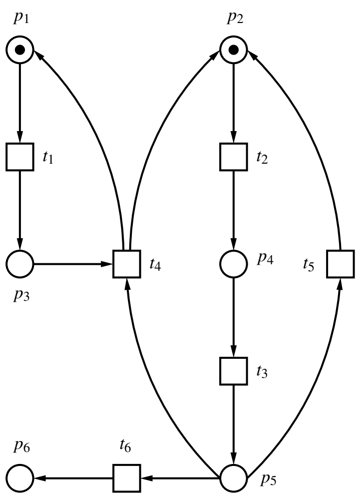
- We consider in this tutorial two simple classes of asynchronous systems but most of the results will carry over to more complicated models of concurrency.
- The two system models considered are:
  - 1-bounded Petri nets
  - Products of labelled transition systems (LTSs)

Petri nets

The class Petri nets we use are called place/transition nets (P/T-nets). A P/T-net is a tuple $N = (P, T, F, W, M_0)$, where

- P is a finite set of places,
- T is a finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
- $W : F \mapsto \mathbb{N} \setminus \{0\}$ is the arc weight mapping, and
- $M_0 : P \mapsto \mathbb{N}$ is the initial marking.

Running Example P/T-net



The running Example

- Places $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$.
- Transitions $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$.
- Flow relation $F = \{(p_1, t_1), (t_1, p_3), (p_2, t_2), (t_2, p_4), (p_4, t_3), (t_3, p_5), (p_3, t_4), (p_5, t_4), (t_4, p_1), (t_4, p_2), (p_5, t_5), (t_5, p_2), (p_5, t_6), (t_6, p_6)\}$.
- Arc weight mapping $W(x, y) = 1$ for all $(x, y) \in F$. We use the convention that only arcs weights $W(x, y) > 1$ are drawn next to the arc (x, y), i.e., the default arc weight is 1.
- Initial marking $M_0 = \{p_1 \mapsto 1,\ p_2 \mapsto 1,\ p_3 \mapsto 0,\ p_4 \mapsto 0,\ p_5 \mapsto 0,\ p_6 \mapsto 0\}$.

Behaviour of P/T-nets

- The state of a P/T-net consist of a *marking* $M : P \mapsto \mathbb{N}$, which tells for each place how many *tokens* (drawn as black dots) it contains.
- The notation $M(p)$ denotes the number of tokens in place p.
- In our running example $M(p) \leq 1$ for all places $p \in P$, i.e., each place contains at most one token. However, this is not required in general.

Behaviour of P/T-nets

- The *preset* of a node $x \in P \cup T$ is denoted by ${}^\bullet x$ and defined to be: ${}^\bullet x = \{y \in P \cup T \mid (y,x) \in F\}$. The preset of a node consist of those nodes from which an arc to x exist. In our running example ${}^\bullet t_4 = \{p_3, p_5\}$.
- The *postset* of a node $x \in P \cup T$ is denoted by $x^\bullet$ and defined to be: $x^\bullet = \{y \in P \cup T \mid (x,y) \in F\}$. The postset of a node consist of those nodes to which an arc from x exist. In our running example $t_4{}^\bullet = \{p_1, p_2\}$.

Enabling of transitions

- A transition $t \in T$ is enabled in marking M, denoted $t \in enabled(M)$, if and only if (iff from now on) for all $p \in {}^\bullet t : M(p) \geq W(p,t)$.
(All places p which are in the preset of t contain at least the number of tokens specified by $W(p,t)$.)

Firing of transitions

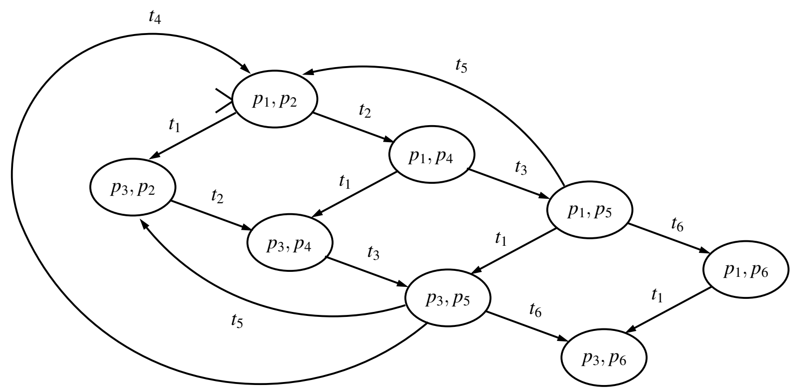
- To simplify definitions, we extend $W(x,y)$ to all pairs $(x,y) \in (P \cup T) \times (T \cup P)$ as follows: if $(x,y) \notin F$ then $W(x,y) = 0$.
- The marking M' reached after firing t, denoted $M' = \mathit{fire}(M,t)$, is defined for all $p \in P$ as:
$M'(p) = M(p) - W(p,t) + W(t,p)$.
(First remove as many tokens as given by $W(p,t)$ from all places in the preset of t, and then add as many tokens for all places in the postset of t as denoted by $W(t,p)$.)

Reachability graph

Reachability graph $G = (V, E, M_0)$ is the graph inductively defined as follows:

- $M_0 \in V$, where M_0 is the initial marking of the net N, and
- if $M \in V$ then for all $t \in enabled(M)$ it holds that $M' = fire(M,t) \in V$ and $(M,t,M') \in E$.

Reachability Graph



Reachability graph (cnt.)

- A place $p \in P$ is defined to be k-bounded iff for all reachable markings $M \in V$ it holds that $M(p) \leq k$.
- A net is defined to be k-bounded if all its places are k-bounded
- In the following we consider how to encode the transition relation $T(s,s')$ for 1-bounded P/T-nets only.

Interleaving Executions

- When the net is 1-bounded, we will use set notation for markings.
- In our running example the initial marking $M_0 = \{p_1, p_2\}$.
- In the initial marking the transition t_2 is enabled and its firing lead to marking $M' = \{p_1, p_4\}$. We denote this by: $\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}$.
- One interleaving execution of length 4 leading to a deadlock; a marking with no enabled transitions is:

$$\{p_1,p_2\}[t_2\rangle\{p_1,p_4\}[t_3\rangle\{p_1,p_5\}[t_6\rangle\{p_1,p_6\}[t_1\rangle\{p_3,p_6\}$$

Conjunctive Normal Form (CNF)

- The SAT solvers mentioned so far require the problem to be mapped into the so called conjunctive normal form (CNF).
- A literal is either either a propositional variable x or its negation $\neg x$. A formula is in conjunctive normal form if it is a conjunction of clauses, where each clause is a disjunction of literals.
- Example: $(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$ is in CNF.
- Using CNF formulas makes the implementation techniques inside SAT solvers simpler which often leads to more efficient implementation techniques.

Constrained Boolean Circuit SAT

- To raise the abstraction level a bit, in this tutorial we will use constrained Boolean circuit SAT instead of CNF.
- They are Boolean circuits where some of the output gates are constrained to *true*, while other can be constrained to *false*.
- The solver now has to find a valuation for the input variables of the circuit to make all the constraints to match the value computed by the circuit.

Boolean Circuit Format

- The format of the circuits used is described in http://www.tcs.hut.fi/~tjunttil/circuits/index.html. The page also contains constrained Boolean circuit front-ends to the zChaff and MiniSAT solvers, which internally convert the Boolean circuits into CNF.

From Circuits to CNF

- The page mentioned above also contains a tool called bc2cnf, with which you can obtain CNF formulas for other CNF based solvers.
- The translation to CNF is based on the Tseitin CNF encoding (see, e.g. page 562 of Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking. Computational Logic 2000: 553-567, LNCS 1861.)
- Tseitin encoding introduces one new variable for each gate of the Boolean circuit, and then encodes the value of that gate with a small equivalence, translated into CNF.

From Circuits to CNF (cnt.)

- An AND gate $x := AND(y,z);$ would become: $x \Leftrightarrow (y \wedge z)$, and translated into CNF would give: $(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$
- An OR gate $x := OR(y,z);$ would become: $x \Leftrightarrow (y \vee z)$, and translated into CNF would give: $(\neg x \vee y \vee z) \wedge (x \vee \neg y) \wedge (x \vee \neg z)$
- A constraint $x := true;$ will contribute to the CNF: x.
- A constraint $x := false;$ will contribute to the CNF: $\neg x$.

Cardinality Gates

- We also use a special gate type called the cardinality gate: $x := [0,1](a_0, a_1, \ldots, a_{m-1});$ which evaluates to true iff at most one of the m input variables $\{a_0, a_1, \ldots, a_{m-1}\}$ is true.
- This can be simulated with m new variables b_i, as follows: $b_0 := false$; for all $1 \leq i \leq m-1$ we have $b_i := b_{i-1} \vee a_{i-1}$; and the final value is obtained by $x := \neg((a_1 \wedge b_1) \vee \cdots \vee (a_{m-1} \wedge b_{m-1}));$.
- There are also other linear size translations for replacing the cardinality gates with ANDs and ORs, the one above is picked for its simplicity.

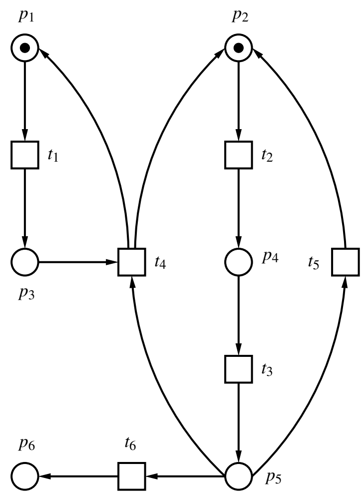
Cardinality Gates (cnt.)

- There is another $O(m^2)$ translation which does not introduce any new variables.
- It seems to have better performance for small values of m in CNF based SAT solvers.
- The use of cardinality gates is a vital ingredient to obtain a small encoding of the transition relation $T(s,s')$ for asynchronous systems.

The Transition Relation Encoding

- The following encoding almost identical to the one in: Keijo Heljanko: Bounded Reachability Checking with Process Semantics. CONCUR 2001: 218-232, LNCS 2154.

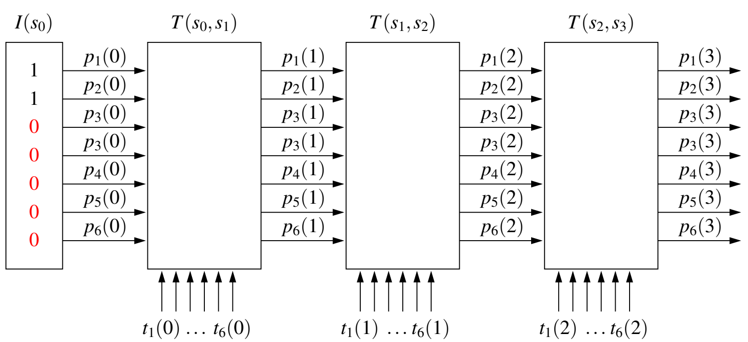
Running Example (recap)



State Variables and Inputs

- We now show how given a 1-bounded P/T-net its transition relation can be encoded into a constrained Boolean circuit.
- The mapping has the following intuition:
  - The state vector bit $p_j(i)$ $(= s_i[j])$ will be true iff in state s_i the place p_j contains a token. For example $p_3(0)$ is the variable corresponding to the place p_3 at the initial state s_0.
  - The Boolean circuit will have one free input variable $t_j(i)$ for each transition $t_j \in T$ and each transition relation instance $0 \leq i \leq k-1$. It $t_j(i)$ is true, then the transition t_j is fired at time i.

P/T-net Transition Relation Unrolling



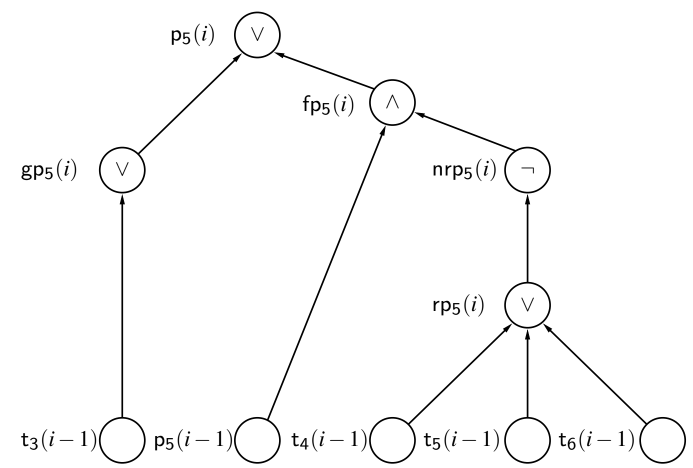
Initial Marking

- Handling the initial marking is easy, and goes as follows:
  - For each $p_j \in P$ such that $M(p) = 1$, set $p_j(0) := \mathit{true}$;
  - For each $p_j \in P$ such that $M(p) = 0$, set $p_j(0) := \mathit{false}$;

Token Updating

- For each place $p_j \in P$ and time $1 \leq i \leq k$ create the following gates:
  - $gp_j(i) := OR(t_1(i-1), t_2(i-1), \ldots, t_l(i-1));$, where $^{\bullet}p_j = \{t_1, t_2, \ldots, t_l\}$. This gate models the generation of a token to p_j.
  - $rp_j(i) := OR(t_1(i-1), t_2(i-1), \ldots, t_l(i-1));$, where $p_j^{\bullet} = \{t_1, t_2, \ldots, t_l\}$. This gate models the removal of a token from p_j.
  - $p_j(i) := gp_j(i) \vee (p_j(i-1) \wedge \neg rp_j(i));$. A token exists in p_j if either a new one was generated, or an old one existed and it was not removed.

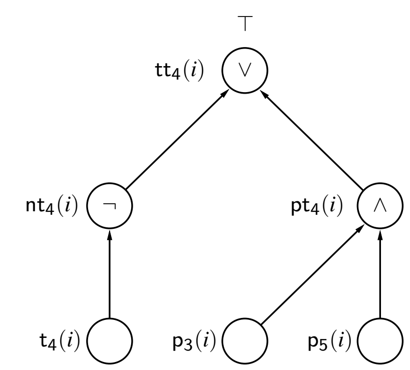
Translation for Place p_5



Transition Enabling

- We should also rule out models where a transition $t_j \in T$ is fired at time $0 \leq i \leq k-1$ without being enabled by using the following gates:
  - Create a gate $pt_j(i) := AND(p_1(i), p_2(i), \ldots, p_l(i));$, where $^{\bullet}t_j = \{p_1, p_2, \ldots, p_l\}$. This gate is true when the transition t_j is enabled.
  - Disallow the firing of disabled transitions with a gate: $tt_j(i) := \neg t_j(i) \vee pt_j(i)$, where $tt_j(i)$ is constrained to *true*. (This is simply a constrained Boolean circuit encoding of $t_j(i) \Rightarrow pt_j(i)$.)

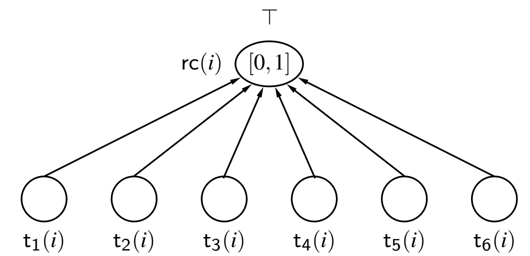
Translation for Transition t_4



Removing Conflicting Transitions

- The encoding so far allows for several transitions to be fired concurrently even if they are in conflict.
- We will disallow this by adding the following gate at each time point $0 \leq i \leq k-1$:
  - $rc(i) := [0,1](t_1(i), t_2(i), \ldots, t_l(i))$, where $T = \{t_1, t_2, \ldots, t_l\}$. We also constrain the gate $rc(i)$ to $true$.
  - The gate $rc(i)$ intuitively removes the possibility of two conflicting transitions to fire at the same time point because at most one transition is allowed to fire at each time point.

Removing Conflicts



Interleaving Semantics

- All parts of the encoding taken together give a constrained Boolean circuit encoding of $T(s, s')$ for the interleaving semantics.
- The interleaving semantics is the standard textbook semantics of P/T-nets where at most one transition is allowed to fire at each time point.
- The size of the encoding is linear in both the size of the input P/T-net and the bound k.

Optional Idling Removal

- The circuit as show above allows no transition to be fired at any index i of the execution.
- It is easy to disallow this by for all $0 \leq i \leq k-1$ introducing a gate: $idle(i) := \neg(OR(t_1(i), t_2(i), \ldots, t_l(i)));$, where $T = \{t_1, t_2, \ldots, t_l\}$. We can now optionally constrain the gate $idle(i)$ to $false$ to remove idling.
- If idling is not removed, the encoding has models corresponding to all executions of length k or less.
- When idling is removed, the encoding has models corresponding to all executions of exactly length k.

Deadlock Detection

- We now assume idling has not been removed, and also drop the assumption that $T(s, s')$ is total.
- Deadlocking executions of length k or less can now be captured by adding the following constraint on the places $p_i(k)$:
- First add the translation of the transition preset gates $pt_j(i)$ also for the index $i = k$.
- Add a gate $dead(k) := \neg(OR(pt_1(k), pt_2(k), \ldots, pt_l(k)))$;, where $T = \{t_1, t_2, \ldots, t_l\}$. Constrain the gate $dead(k)$ to *true* to capture executions leading to a state where no transition is enabled: a deadlock state.

Deadlock Checking Demo (1/3)

```
$ cat running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 4
semantics = "interleaving"

$ 1b-pn-bmc < running.net | bczchaff | cex-print
{p1, p2}[t1>{p2, p3}[t2>{p3, p4}[t3>{p3, p5}[t6>{p3, p6}
```


Deadlock Checking Demo (2/3)

```
$ 1b-pn-bmc < running.net | bczchaff -v
Parsing from stdin
The circuit has 196 gates
The input gates are: t6_3 t3_3 t2_3 t5_3 t1_3 t4_3 t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1
t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 138 gates and 153 edges after simplification
The circuit has 83 gates and 134 edges after sharing
The circuit has 75 gates and 92 edges after simplification
The circuit has 59 gates and 89 edges after sharing
The circuit has 54 gates and 66 edges after simplification
The circuit has 48 gates and 65 edges after sharing
The circuit has 48 gates and 65 edges after simplification
The circuit has 48 gates and 65 edges after sharing
The circuit has 56 gates after normalization
The circuit has 56 gates and 71 edges after simplification
The circuit has 52 gates and 71 edges after sharing
The circuit has 52 gates and 71 edges after simplification
The circuit has 52 gates and 71 edges after sharing
The max-min height of the circuit is 2
The max-max height of the circuit is 4
The circuit has 46 relevant gates
The circuit has 10 relevant input gates
The cnf has 37 variables and 96 clauses
```


Deadlock Checking Demo (3/3)

```
Executing zchaff...

Max Decision Level 1
Num. of Decisions 2
Original Num Clauses 96
Original Num Literals 200
Added Conflict Clauses 0
Added Conflict Literals 0
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 37
~live_4 ~gp4_4 ~t2_3 ~gp5_4 ~t3_3 ~gp1_4 ~t4_3 ~t5_3 ~gp1_2 ~t4_1 ~t5_1 ~t5_0 ~gp1_1
~t4_0 ~p5_4 ~p4_4 ~p2_4 ~p1_4 ~p6_2 ~gp6_2 ~t6_1 ~p6_1 ~gp6_1 ~t6_0 ~p5_1 ~gp5_1
~t3_0 ~p6_0 ~p5_0 ~p4_0 ~p3_0 ~gp1_3 ~t4_2 ~t5_2 ~gp2_1 ~gp2_2 ~gp2_4 ~gp4_3 ~t2_2
~p4_3 ~p2_3 ~gp2_3 rc1 rc2 gate13 gate15 gate16 gate11 gate10 gate9 rc0 gate5 gate4 gate3
gate2 gate1 gate0 p2_0 p1_0 gate19 gate20 gate21 gate22 rc3 gate23 gate18 p6_4 gp6_4 t6_3
p3_4 ~gp3_4 ~t1_3 gate17 gate14 gate12 p5_3 ~p6_3 ~gp6_3 ~t6_2 gp5_3 t3_2 p3_3 ~p1_3
~gp3_3 ~t1_2 gate8 gate7 gate6 p4_2 ~p5_2 ~gp5_2 ~t3_1 p3_2 ~p2_2 gp4_2 t2_1 ~p1_2
~gp3_2 ~t1_1 p2_1 ~p4_1 ~gp4_1 ~t2_0 ~p1_1 p3_1 gp3_1 t1_0
Satisfiable
```

Step Semantics

- An old well known semantics from the theory of Petri nets is the so called step semantics.
- This is not the same as maximal step semantics.
- The idea is the following: Instead of firing a single enabled transition at each marking M, we can fire a step: a set of enabled transitions $S \subseteq enabled(M)$ at a time point provided they are all pairwise concurrent:
  - For all pairs of distinct transitions $t, t' \in S$ it holds that ${}^\bullet t \cap {}^\bullet t' = \emptyset$.
  - Note: It is straightforward to prove that because we only consider 1-bounded P/T-nets here that actually also $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$ holds.

Step Reachability graph

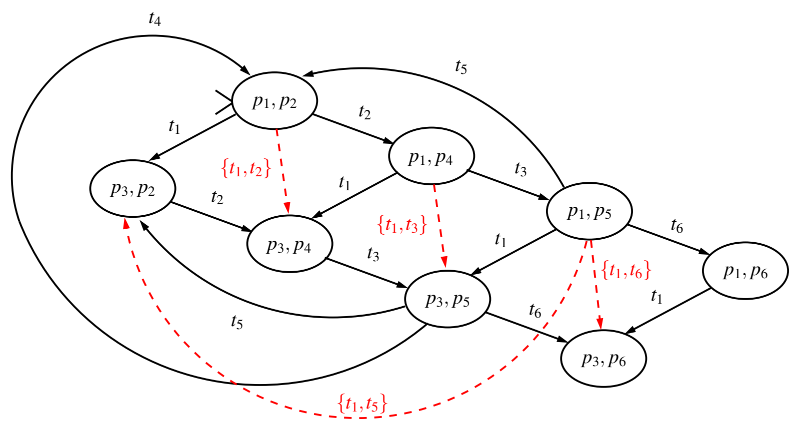
Step reachability graph $G_s = (V, E, M_0)$ is the graph inductively defined as follows:

- $M_0 \in V$, where M_0 is the initial marking of the net N, and
- if $M \in V$ then for all $S \subseteq enabled(M)$ such that S is a step it holds that $M' = fire(M, S) \in V$ and $(M, S, M') \in E$.

Some Properties of Steps

- If a set of transitions $S = \{t_1, t_2, \ldots, t_l\}$ is step enabled in M, then all the $l!$ interleaving executions obtained by sequentialising S in all orders are enabled interleaving executions in M, and they all lead to the same final state.
- An intuition why this is the case: Because $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$, the transitions t and t' happen "in different parts of the system", and thus cannot influence each other in any way.
- Thus $fire(M, S)$ from the previous slide can be defined as: $fire(\ldots(fire(fire(M, t_1), t_2), \ldots, t_l)$.

Step Reachability Graph



Properties Steps Graphs

- Because all singleton sets are also steps, the (interleaving) reachability graph is always a subgraph of the step reachability graph.
- Because the final state reached after firing a step is the final state of every interleaving of the step, no new reachable states have been introduced.
- The reachability diameter of the system is in the worst case as big as in the interleaving case.
- In the best case the interleaving diameter has become smaller, because a step with o transitions has to be simulated with o time steps in the interleaving reachability graph.

Running Example: Steps

- We extend the notation $M[t\rangle M'$ to also denote the firing of steps $M[S\rangle M'$.
- In the running example one step executions of length 3 leading to the deadlock marking $\{p_3, p_6\}$ is:

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

- Recall that the shortest execution to $\{p_3, p_6\}$ was of length 4 in the interleaving reachability graph.
- $\Rightarrow$ Using step semantics allows one to sometimes detect errors with smaller bounds.

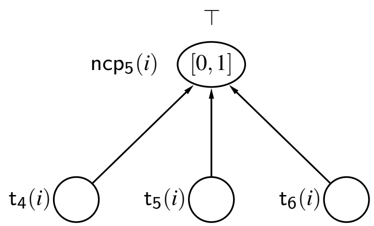
Encoding the Step Semantics

- It is easy to modify the interleaving transition relation encoding to encode step semantics instead.
- The only thing we have to remove is the encoding of gate $rc(i)$, which restricts the number of fired transitions to at most one as required by the interleaving case.
- A set of constraints has to be added to remove the possibility of two conflicting transitions to fire in the same step.

Steps: Removing Conflicts

- For each place $p_j \in P$ and each step $0 \leq i \leq k-1$ we will add the following gate which disallows concurrent firing of transitions which are not concurrent due to both having the place p_j in their preset:
- $ncp_j(i) := [0,1](t_1(i), t_2(i), \ldots, t_l(i))$, where $p_j{}^\bullet = \{t_1, t_2, \ldots, t_l\}$. We also constrain the gate $ncp_j(i)$ to $true$.

Steps: Removing Conflicts wrt. p_5



Demo with Step Semantics (1/3)

```
$ cat step-running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 3
semantics = "step"

$ 1b-pn-bmc < step-running.net | bczchaff | cex-print
{p1, p2}[t2>{p1, p4}[t1, t3>{p3, p5}[t6>{p3, p6}
```


Demo with Step Semantics (2/3)

```
$ 1b-pn-bmc < step-running.net | bczchaff -v
Parsing from stdin
The circuit has 149 gates
The input gates are: t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1 t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 99 gates and 92 edges after simplification
The circuit has 48 gates and 74 edges after sharing
The circuit has 39 gates and 33 edges after simplification
The circuit has 27 gates and 32 edges after sharing
The circuit has 24 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The circuit has 14 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The circuit has 14 gates after normalization
The circuit has 14 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The max-min height of the circuit is 2
The max-max height of the circuit is 3
The circuit has 10 relevant gates
The circuit has 3 relevant input gates
The cnf has 8 variables and 15 clauses
```


Demo with Step Semantics (3/3)

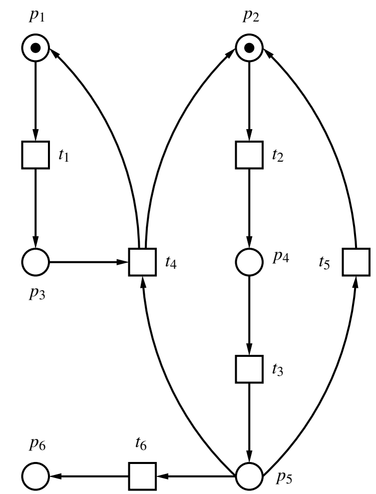
```
Executing zchaff...

Max Decision Level 2
Num. of Decisions 3
Original Num Clauses 15
Original Num Literals 31
Added Conflict Clauses 0
Added Conflict Literals 0
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 8
˜p2_1 ˜p2_2 ˜p4_2 ˜t2_1 ˜gp4_2 ˜gp2_3 ˜gp2_2 ˜gp2_1 ˜p3_0 ˜p4_0 ˜p5_0 ˜p6_0 ˜t3_0 ˜gp5_1 ˜p5_1 ˜t6_0
˜gp6_1 ˜p6_1 ˜t6_1 ˜gp6_2 ˜p6_2 ˜p1_3 ˜p2_3 ˜p4_3 ˜p5_3 ˜t4_0 ˜gp1_1 ˜t5_0 ˜t5_1 ˜t2_2 ˜gp4_3 ˜t3_2
˜gp5_3˜t4_1 ˜gp1_2 ˜t4_2 ˜gp1_3 ˜t5_2 ˜live_3 t2_0 gp4_1 p4_1 t3_1 gp5_2 p5_2 gate7 gate8 t6_2
gp6_3 p6_3 gate16 gate15 gate14 gate13 ncp5_2 p1_0 p2_0 gate0 gate1 gate2 gate3 gate4 gate5
gate9 gate10 gate11 ncp5_0ncp5_1 gate17 gate12 p3_3 ˜gp3_3 ˜t1_2 gate6 p3_2 ˜p1_2 gp3_2 t1_1
p1_1 ˜p3_1 ˜gp3_1 ˜t1_0
Satisfiable
```

Interleaving vs. Steps

- We have not yet found a domain where the interleaving encoding would be superior in performance to the step encoding.
- Quite often even small reductions in the required bound translate to large performance differences.
- The step encoding also is more "local" than the interleaving encoding:
  - Parts of the system which do not share resources are never linked together as done by the $rc(i)$ gate in the interleaving case.
  - This might have SAT performance implications.

Running Example (recap2)



Process Semantics

- In our running example there are three step executions of length 3 leading to the deadlock marking $\{p_3, p_6\}$:

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_3\rangle\{p_1, p_5\}[t_1, t_6\rangle\{p_3, p_6\}$$

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

$$\{p_1, p_2\}[t_1, t_2\rangle\{p_3, p_4\}[t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

- Intuitively they all correspond a concurrent execution where "the component on the left" executes t_1, and "the component on the right" executes the sequence t_2, t_3, t_6.

Process Semantics (cnt.)

- Can we somehow pick a unique canonical representative of such "concurrent" behaviour, and thus reduce the number of different executions the SAT solver has to consider?
- The answer turns out to be positive. The resulting semantics will be called process semantics.
- There is even a compact SAT encoding to capture the process semantics!

The Process Normal Form

- A step execution $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{k-1}[S_{k-1}\rangle M_k$ is in process normal form iff for every index $i \geq 1$ and every transition $t_j \in S_i$ it holds that:
  - There is some transition $t' \in S_{i-1}$ such that $t'^{\bullet} \cap {}^{\bullet}t_j \neq \emptyset$.

Process Normal Form Intuition

- Intuitively the above means: In a step execution in process normal form each transition is executed at the earliest time moment all its tokens are available.
- In the SAT encoding setting this means that a transition should be enabled only if one of its tokens has been generated in the previous step.
- Thus the process execution for the example is:

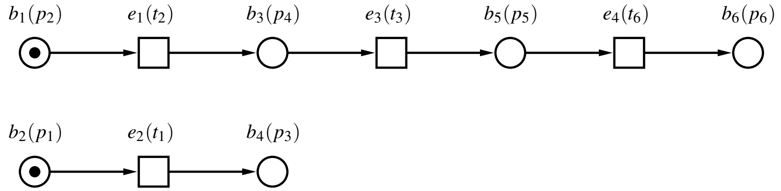
$$\{p_1, p_2\}[t_1, t_2\rangle\{p_3, p_4\}[t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

Normalising a Step Execution

- By repeatedly running the following simple algorithm each step execution $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{k-1}[S_{k-1}\rangle M_k$ can be converted into a process executions of at most the same length and leading to the same final state:
  - Take a transition $t_j \in S_i$ which violates the process condition.
  - Remove t_j from S_i and add it to S_{i-1}.
- Proof is simple, one has to show that: (i) t_j is enabled already in S_{i-1}, and (ii) S_{i-1} contains no transitions in conflict with t_j.

Process

As a graphical presentation of the process we can again use a P/T-net:



The step executions in process normal form correspond to slicing the net one level at a time starting from the left.

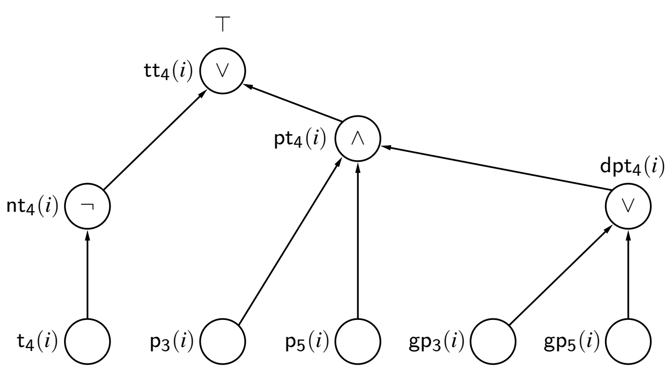
Properties of Processes

- Each state of the system is reachable by a process execution that is among the shortest step executions to reach that state.
- Thus the set of reachable states is preserved.
- Furthermore, the process reachability diameter is always as small as the step reachability diameter.
- There are at most as many process executions of length k as there are interleaving executions of length k.
- There can be exponentially more step and interleaving executions of length k than there are process executions.

Encoding Process Semantics

- Take the encoding for the step semantics but change the transition enabling gate definition for all $i > 0$:
- For each transition $t_j \in T$ and time $1 \leq i \leq k-1$ use the following gates (for $i = 0$ use the step version):
  - Create a gate $pt_j(i) := AND(p_1(i), p_2(i), \ldots, p_l(i), OR(gp_1(i), gp_2(i), \ldots, gp_l(i)));$, where $^\bullet t_j = \{p_1, p_2, \ldots, p_l\}$. This gate is true when the transition t_j is enabled, and at least one of the tokens has been freshly generated.
  - Add gate: $tt_j(i) := \neg t_j(i) \vee pt_j(i)$, and constrain it to *true*.

Process Translation for t_4



Demo with Process Semantics (1/3)

```
$ cat process-running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 3
semantics = "process"

$ lb-pn-bmc < process-running.net | bczchaff | ./cex-print
{p1, p2}[t1, t2>{p3, p4}[t3>{p3, p5}[t6>{p3, p6}
```

Demo with Process Semantics (2/3)

```
$ 1b-pn-bmc < process-running.net | bczchaff -v
Parsing from stdin
The circuit has 161 gates
The input gates are: t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1 t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 101 gates and 88 edges after simplification
The circuit has 44 gates and 70 edges after sharing
The circuit has 30 gates and 11 edges after simplification
The circuit has 10 gates and 10 edges after sharing
The circuit has 8 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The circuit has 2 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The circuit has 2 gates after normalization
The circuit has 2 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The max-min height of the circuit is 0
The max-max height of the circuit is 0
The circuit has 0 relevant gates
```

Note that with the more constrained process encoding, the preprocessing already solves the circuit.

Demo with Process Semantics (3/3)

```
˜p1_1 ˜p2_1 ˜p2_2 ˜p1_2 ˜gp2_2 ˜gp2_1 ˜gp2_3 ˜t2_1 ˜gp4_2 ˜p4_2 ˜p3_0 ˜p4_0 ˜p5_0 ˜p6_0 ˜t3_0
˜gp5_1 ˜p5_1 ˜t6_0 ˜gp6_1 ˜p6_1 ˜t6_1 ˜gp6_2 ˜p6_2 ˜t1_2 ˜gp3_3 ˜p1_3 ˜p2_3 ˜p4_3 ˜p5_3 ˜t5_0
˜t4_0 ˜gp1_1 ˜t5_1 ˜t2_2 ˜gp4_3 ˜t3_2 ˜gp5_3 ˜t4_1 ˜gp1_2 ˜t1_1 ˜gp3_2 ˜t4_2 ˜gp1_3 ˜t5_2 ˜live_3
t3_1 gp5_2 p5_2 gate8 t6_2 gp6_3 p6_3 gate16 p3_3 p3_2 gate15 gate14 gate13 p1_0 p2_0 gate0
gate1 gate2 gate3 gate4 gate5 gate6 gate9 gate10 gate11 gate12 ncp5_2 p4_1 gp4_1 t2_0 gate7
ncp5_0 ncp5_1 p3_1 gp3_1 t1_0 gate17
Satisfiable
```

No need to invoke zChaff, just output the solution.

Steps vs. Processes

- Unfortunately there is some bad news: processes are not always faster than steps with the latest SAT solvers such as zChaff and Siege. (Cause unknown.)
- Often a polynomial time preprocessing algorithm is used to compute the earliest time each transition can fire or a place can become marked.
- This allows for simplification of the BMC encoding by introducing constant values for variables.
- Process semantics can be used as a better polynomial time preprocessing step as it can prove for more transitions that they can never be enabled at certain time points in process executions.

Steps vs. Processes (cnt.)

- The transition relation for steps can be represented as: $T(s_i, i_i, s_{i+1})$, where i_i is the current input vector (in the running example, the set of transitions to be fired in step S_i: $i_i = \langle t_1(i), t_2(i), \ldots, t_6(i) \rangle$).
- The transition relation for processes can be represented as: $T(s_i, i_{i-1}, i_i, s_{i+1})$, where i_i is the current input vector (step S_i) and i_{i-1} is the previous input vector (step S_{i-1}).

History Dependence

- Thus the process semantics semantics has a history dependent transition relation.
- Another way to present this is to use "three valued tokens", a place can either contain: no tokens, contain a freshly generated token, or contain an old token.
- This makes the use of process semantics unattractive in a BDD model checking setting, as the number of state bits needed to represent the state vector grows.

Processes and Temporal Induction

- It follows from Theorem 17 in Toni Jussila's Doctoral dissertation that the *SimplePath* constraint used in the temporal induction (k-induction) method (to be presented later in this tutorial) can treat old tokens and fresh tokens alike.
- Thus for reachability from the initial state the so called recurrence diameter (to be defined later) for processes is never worse for processes than for steps, but can sometimes even be better.

Model Checking LTL-X

- One can also do model checking of the temporal logic LTL-X with step semantics. (Extension to allow also processes to be used is work in progress.)
- LTL-X is the subset of LTL where the next-time operator X has been removed. This restriction of the logic is often done also with other partial order methods.
- First one has to identify all visible transitions of the net, which can modify the truth value of some atomic proposition in the formula.

Model Checking LTL-X (cnt.)

- All of the visible transitions are made to conflict with each other by adding a new marked place v to the net, and adding a bidirectional arc from each visible transition to v.
- If the is deadlock free, one can additionally require that the last step S_k is non-empty to disallow illegal counterexamples by infinite sequences of idling.
- If the net can deadlock, the solution is more subtle, and we refer to our paper on the subject: Keijo Heljanko, Ilkka Niemelä: Bounded LTL model checking with stable models. TPLP 3(4-5): 519-550 (2003), Cambridge University Press.

Steps and AI Planning

- In AI planning papers a step like optimisation to decrease the needed bounds was already used. Henry A. Kautz, Bart Selman: Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. AAAI/IAAI, Vol. 2 1996: 1194-1201.
- Rintanen et al. discuss SAT encodings of the step semantics and its generalisations for AI planning in: Jussi Rintanen, Keijo Heljanko, Ilkka Niemelä: Parallel Encodings of Classical Planning as Satisfiability. JELIA 2004: 307-319, LNCS 3229.
- The sizes of the encodings mentioned above are quadratic in the number of planning operator instances.

Steps and AI Planning

- There is a SAT encoding that is linear in the number of planning operator instances described in: Rintanen, J., Heljanko, K., and Niemelä, I.: Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. Technical report 216, Institute of Computer Science at Freiburg University, 2005.
- The paper mentioned above contains state-of-the-art CNF translations for AI planning by Jussi Rintanen. The STRIPS planning formalism used is a generalisation of 1-bounded P/T-nets. (A journal submission of an extended version of the paper above been accepted.)

Other Semantics for BMC

- The paper by Rintanen et al. also contains a generalisation of step executions, which allows a set of transitions S to be fired as a step if at least one interleaving of S is executable.
- Other new and efficient non-standard execution semantics for BMC of asynchronous systems have been presented in: Toni Jussila. On bounded model checking of asynchronous systems. Research Report A97, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, October 2005. Doctoral dissertation.

Other Semantics for BMC

- One more approach is presented in: Shougo Ogata, Tatsuhiro Tsuchiya, Tohru Kikuno: SAT-Based Verification of Safe Petri Nets. ATVA 2004: 79-92, LNCS 3299.
- All of the above semantics preserve the set of reachable states.

Process References

- The process normal form we use is basically the Foata normal form from the theory of Mazurkiewicz traces. See for example: Diekert, V. and Métivier, Y.: Partial Commutation and Traces, Handbook of formal languages, Vol. 3, pp. 457–534, Springer, 1997.
- For more on process semantics see for example: Best, E. and Fernández, C.: Nonsequential Processes: A Petri Net View, EATCS monographs on Theoretical Computer Science, Vol. 13, Springer, 1988.

Steps and Processes for LTSs

- Next we describe how to transfer the step and process semantics to systems composed of a synchronisation of labelled transition systems (LTSs).
- The encoding to be presented has been published in: Toni Jussila, Keijo Heljanko, Ilkka Niemelä: BMC via on-the-fly determinization. STTT 7(2): 89-101 (2005).

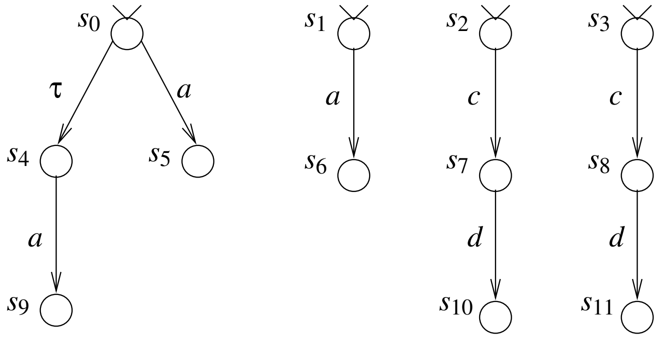
Intuition: LTS Semantics

- We use the standard synchronisation construction for LTSs (see the paper mentioned in the previous slide for details): The system consists of n LTSs $L_1, L_2, \ldots, L_n$ composed as $L = L_1 || L_2 || \cdots || L_n$.
- Each LTS has its own alphabet. The system L can make a move with a letter a iff every LTS with a in its alphabet is able to perform it.
- When a is performed, every LTS with a in its alphabet moves, while the others do not change their state.
- In addition, each LTS can make local τ-labelled moves at will, during which the other components of the system do not change their state.

Alternative Semantics

- Next we show by using a running example what the state spaces induced by the presented alternative semantics for LTSs are.
- Thanks to Toni Jussila for allowing the use of Figures from his Thesis in the following slides.
  Toni Jussila.
  On bounded model checking of asynchronous systems. Research Report A97, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, October 2005. Doctoral dissertation.

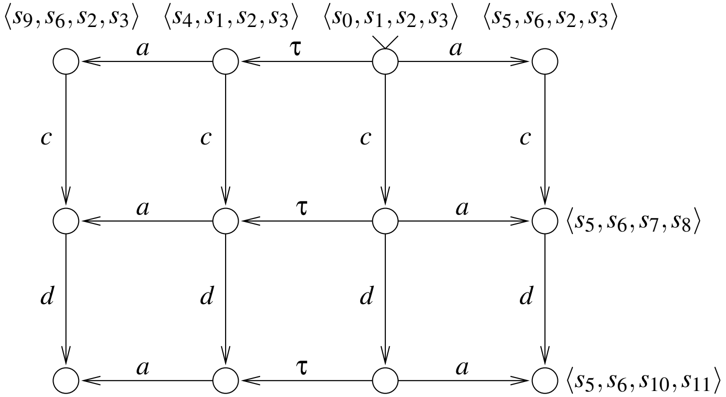
LTSs: Running Example



The complete system is $L = L_1 || L_2 || L_3 || L_4$.

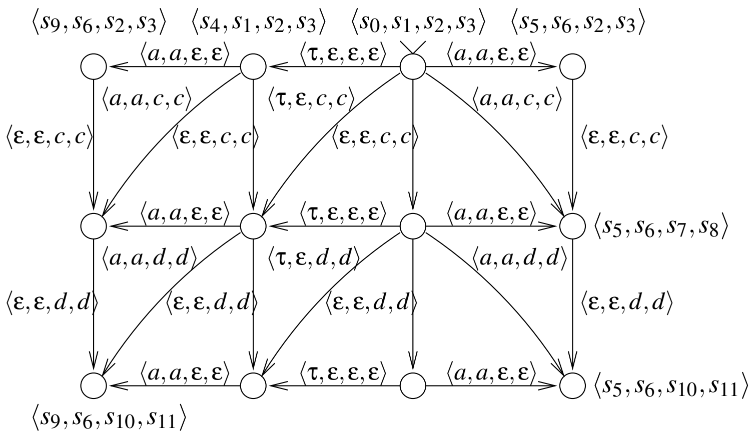
LTSs: Interleaving Semantics

The interleaving semantics is as expected:



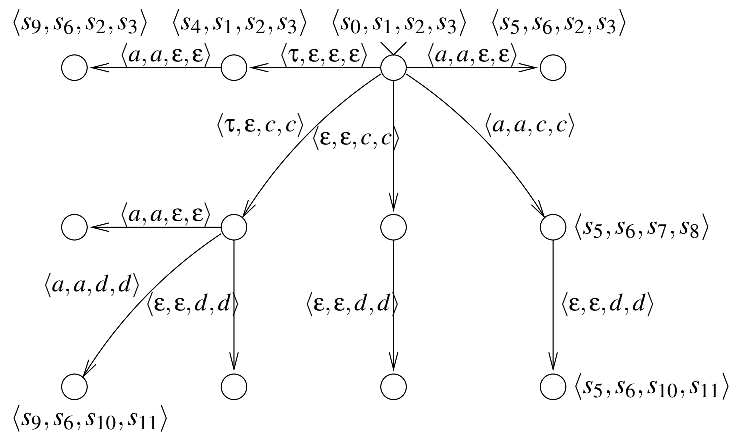
LTSs: Step Semantics

In step semantics two synchronisations are independent if they occur in disjoint sets of LTSs:



LTSs: Process Semantics

In the process case a synchronisation can happen at step S_i iff at least one participant of it was active at step S_{i-1}:

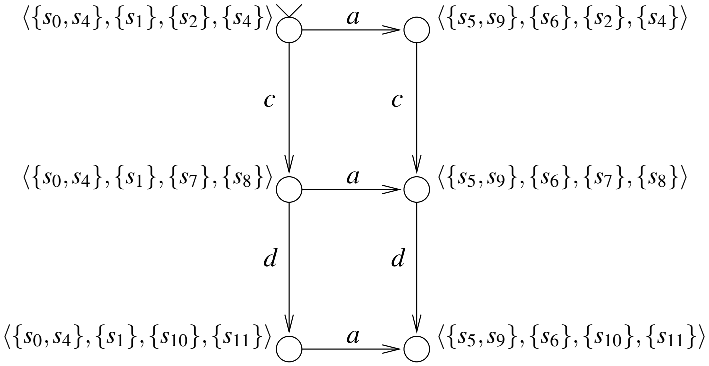


Symbolic Subset Construction

- The FSA subset construction can be used to determinise nondeterministic state machines symbolically inside BMC.
- The tricky part is the correct handling of the τ-moves.
- By doing this, the number of executions through the statespace of the system is further reduced.
- It has also other applications: One can, for example, create a BMC encoding that accepts all words not in the language of L. This has uses, for example, in refinement checking of two products of LTSs.

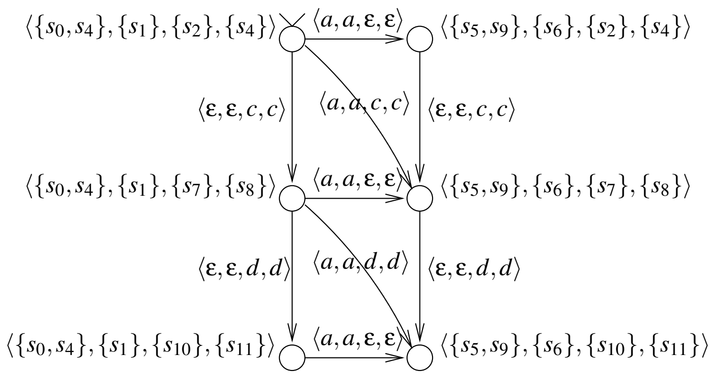
LTSs: Determinised Interleaving

Interleaving combined with determinising each component symbolically during BMC:



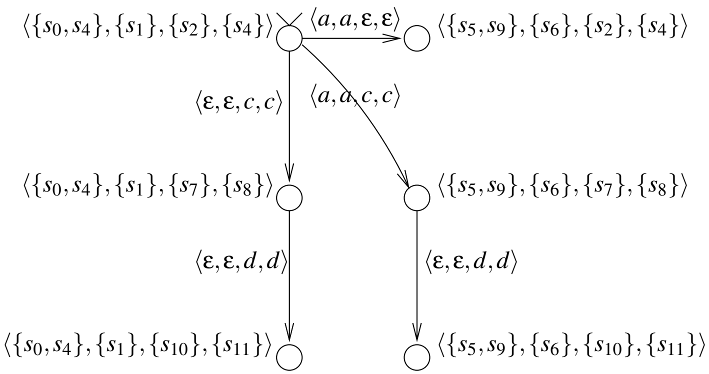
LTSs: Determinised Step

Steps combined with determinising each component symbolically during BMC:



LTSs: Determinised Process

Processes combined with determinising each component symbolically during BMC:



Determinisation Discussion

- The determinised versions preserve the language over the alphabet of L.
- The τ-moves do not contribute to the bound needed.
- The input variables needed for the determinised versions are: one input variable for each symbol in the alphabet of L per time step.
- Thus the determinised version of the encoding can be very attactive for small alphabets.
- The reachability of local states is preserved.

Determinisation Discussion (cnt.)

- Global state predicates such as deadlock freedom need to be evaluated by guessing a representative final state from the set of final states reached, see the paper for details.
- Explicit state determinisation of components can be a viable alternative in many cases, however, there is the potential for an exponential size blowup.

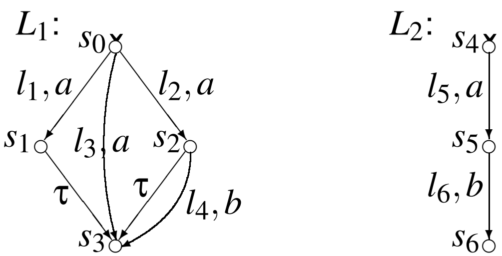
LTS Encoding

- In the following we present by using a running example few of the main ideas of the encoding of the determinised process executions.
- See the STTT paper or Toni Jussila's Thesis for more details.

Technical Restriction

- For technical reasons, we add the following constraint on component LTSs:
  - If a component LTS contains a loop consisting of τ-transitions only, the loop must be a self-loop. (Easy to assure by pre-processing LTSs to ones fulfilling the condition using a linear-time algorithm based on Tarjan's MSCC algorithm.)
- Without the above restriction the encoding becomes unsound! (It would become a cyclic circuit.)
- Also the sets of initial states are extended to contain all states reachable from the initial state of each component by using only τ-moves.

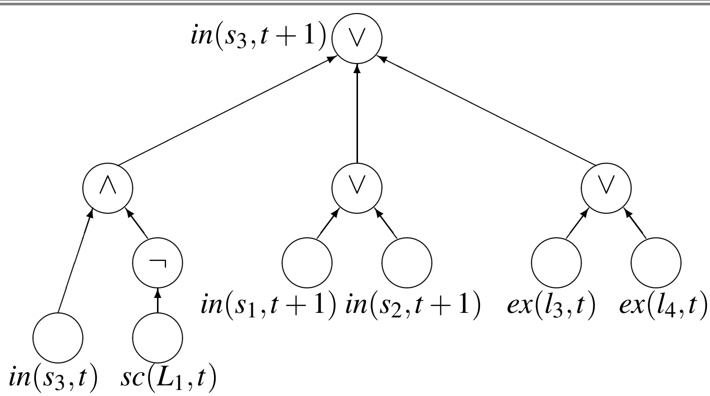
Running Example for LTS Encoding



Translation Predicates

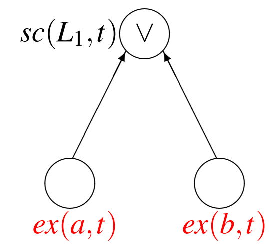
Gate	Description
$ex(a,t)$	Action a is executed at time t, input gate.
$in(s,t)$	Execution is in state s at time t.
$sc(L,t)$	Component L is scheduled at time t.
$ex(l,t)$	Transition l is executed at time t.
$uv(L,t)$	Unique visible transition from L at time t.
$enok(a,t)$	Execution of action a implies that it is enabled at time t.
$en(a,t)$	Action a is enabled at time t.

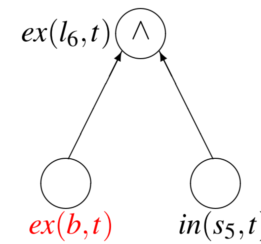
Progress of Control Flow



- We stay in a state if the component is not scheduled. A state can be entered either by τ-edges or by firing of incoming transitions.

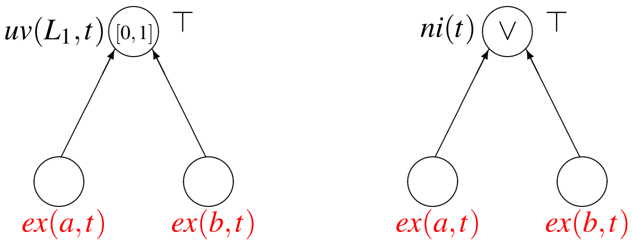
Scheduling (left), Execution (right)





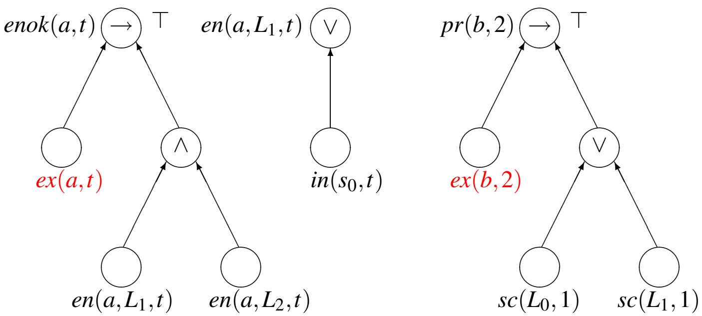
- A component is scheduled iff one of the actions in its alphabet fires.
- A transition is always executed if we are in its source state and the action it is labeled with is fired.

Unique visible (left), Noidle (right)



- The $uv(L_1,t)$ gate disallows more than one action from the alphabet of L_1 at each step.
- The use of non-idling constraint $ni(t)$ is a slight variation to the P/T-net encoding. Here we disallow idling time steps.

Enabledness, Synchronisation



- The two circuits on the left ensure that all components are able to perform the synchronisation on action a. The circuit on the right enforces the process constraint when synchronising on action b.

Conclusions of Tutorial part 1

- Bounded model checking (BMC) is an efficient way of implementing *symbolic model checking*.
- It alleviates the state explosion by representing the state space implicitly as a propositional formula.
- It leverages efficient SAT-solver technology.
- The choice between different transition relation encodings has been often overlooked in BMC literature.
- The performance differences between different transition relation encodings are very significant, at least for asynchronous systems BMC.