Semantic Equivalences and the Verification of Infinite-State Systems

Richard Mayr

Department of Computer Science
Albert-Ludwigs-University Freiburg
Germany

Formal Verification

Proving the correctness of a given hardware- or software system w.r.t. a given specification, or finding the errors in it.

Specification can be described by

- Specification languages
- Temporal logic formulae
- Abstract prototype systems

The goal:

- If the system is correct, then prove it (automatically or semi-automatically).
- If the system is not correct then explain why. Ideally by a short counterexample.

Verification Methods

- Semiautomatic methods
  - Manual or semiautomatic abstraction techniques. Reduction to problems which automatic methods can handle.
  - Theorem provers (e.g., 'Isabelle', PVS).
- Automatic methods
  - Reachability analysis
  - Temporal logics and model checking (e.g., SPIN, SMV).
  - Semantic equivalence checking
  - Semantic preorder checking (i.e., $A \sqsubseteq B$, B implements A).
    - Simulation preorder
    - Trace preorder

Formal Verification Methods

Finite-state systems:

- Theory relatively well understood.
- Problems with handling very large state spaces. Techniques used:
  - BDDs
  - minimization (preserving interesting properties)
  - symmetry reduction
  - PO-reduction
  - search heuristics

Many software systems are infinite-state. How to handle them ?

What makes systems infinite ?

- Unbounded recursion:
  Abstract models: pushdown automata, context-free processes

- Process creation and concurrency:
  Abstract models: Petri nets, Basic Parallel Processes, PA-processes

- Counters:
  Abstract models: counter machines (reversal-bounded, flat, 1-counter)

- Buffers:
  Abstract models: FIFO-channel systems, Lossy FIFO-channel systems

- Real time:
  Abstract models: Timed automata.

- Unbounded data structures:
  Abstract models: Extended automata

Model Checking and Related Methods

- Reachability analysis:
  Symbolic representations of (infinite) sets of states. Algorithms for computing
  - Pre^* (set of predecessors) of a given set of states
  - $Post^*$ (set of successors) of a given set of states

  $\Longrightarrow$ Useful for checking safety-properties.

- Model checking with branching-time temporal logics:
  EF, EG, CTL, CTL*, modal μ-calculus.

- Model checking with linear-time temporal logics:
  LTL, linear-time μ-calculus.

- Semantic equivalence/preorder checking of abstract specification and more refined model. E.g., simulation, bisimulation.

Highlights of my own work

- Process Rewrite Systems [Mayr, 1997].
  - A unified formalism for describing many classes of systems by term rewriting.
  - Strictly more general than Petri nets and pushdown automata.
  - Decidable reachability problem.

- Temporal logic EF
  - Decidability of model checking for process classes PA and PAD [Mayr, 1997,1998].
  - Construction of characteristic EF-formulae for finite-state systems w.r.t. strong and weak bisimulation [Jančar, Kučera, Mayr, 1998].

- Undecidability of boundedness for lossy counter machines [Mayr, 2000].
  Very general undecidability result with consequences for lossy FIFO-channel systems, transfer Petri nets, parameterized systems, etc.

More Highlights

- Undecidability of weak bisimulation equivalence for 1-counter processes [Mayr, 2003].
  - Weak bisimilarity undecidable for Petri nets with just 1 unbounded place.
  - Thus undecidable for almost all classes of infinite-state systems.

- Polynomial time algorithm for checking strong/weak bisimulation equivalence between context-free processes and finite-state systems [Kučera, Mayr, 1999].
  - Works by construction of the bisimulation-base, a compact representation of the whole infinite equivalence relation.

- Why simulation is harder than bisimulation [Kučera, Mayr, 2002].

- Applied verification: A scalable automatic method for analyzing buffer overflow conditions in UML RT and Promela models. $\longrightarrow$ IBOC-tool.

Semantic Equivalences

Consider systems/states as equivalent if they have the same behavior/properties.

What is the same behavior ? $\longrightarrow$ Notions of semantic equivalence.

Does implementation A really implement specification B ?
$\longrightarrow$ Notions of semantic preorder.

Central notions: Simulation and Bisimulation

- Definition
- Game theoretic characterization
- Logical characterization

General Model

Labeled transition graph:

$$(V, \xrightarrow{a}, Act)$$

V	Set of vertices. (Vertices interpreted as states.)
Act	Set of atomic actions.
$\xrightarrow{a} \subseteq V \times Act \times V$	Directed edges, labeled with atomic actions.

Consider semantic equivalences $R \subseteq V \times V$ in a single transition graph.

States of different graphs can be compared by putting the graphs 'side-by-side' and considering them as a single graph.

Semantic Equivalences: Definition

Simulation:

A relation $R \subseteq V \times V$ is a simulation if
whenever $(X, Y) \in R$ then for each $a \in Act$:
$\forall X'$ with $X \xrightarrow{a} X' \quad \exists Y'$ with $Y \xrightarrow{a} Y'$ and $(X', Y') \in R$.

Simulation preorder is the largest simulation relation, denoted by $\sqsubseteq$.

Simulation equivalence $\simeq$ is defined as $\sqsubseteq \cap \sqsupseteq$

Bisimulation:

A relation R is a bisimulation iff it is a symmetric simulation.

The largest bisimulation is an equivalence, denoted by $\sim$.

Simulation Games

Simulation game between two players: Attacker and Defender.

Initial configuration: Pair of states (X, Y).

Attacker wants to show $X \not\sqsubseteq Y$.
Defender wants to show $X \sqsubseteq Y$.

In every round of the game:
Attacker makes a move on the left side, e.g., $X \xrightarrow{a} X'$.
Defender must respond by a move on the right side with the same action, e.g., $Y \xrightarrow{a} Y'$.
The new configuration is (X', Y'). $\longrightarrow$ Next round.

If one player cannot move then the other player wins. The defender wins every infinite game.

Theorem: $X \sqsubseteq Y$ iff the defender has a winning strategy.

Bisimulation Games

Bisimulation game between two players: Attacker and Defender.

Initial configuration (X, Y).

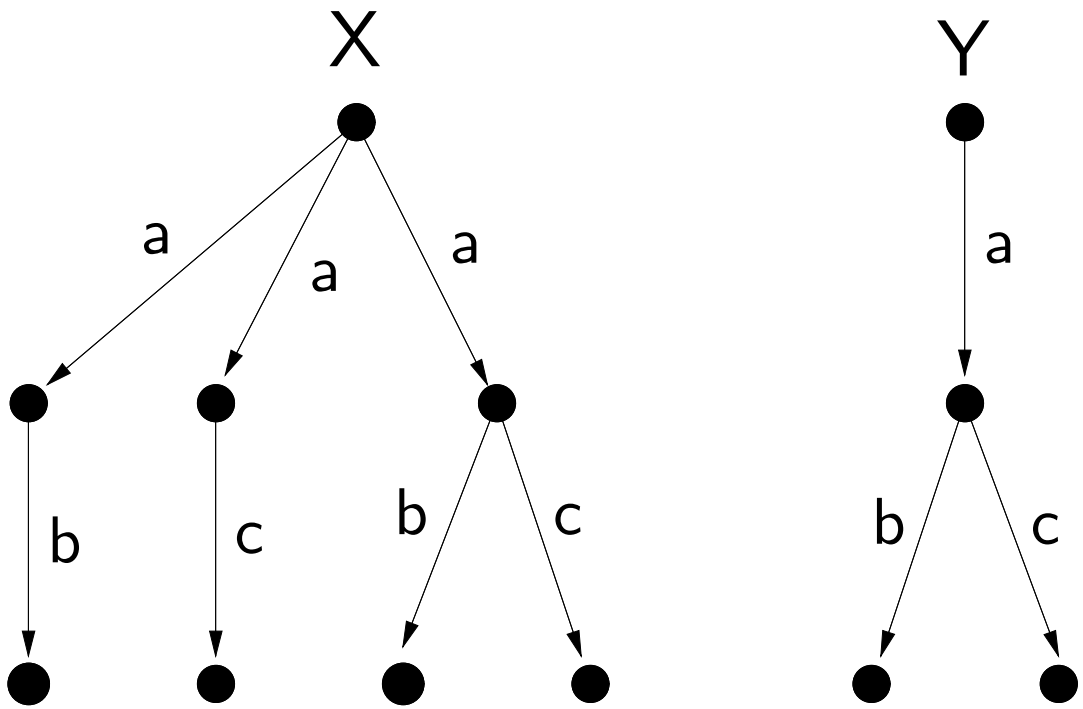
Attacker wants to show $X \not\sim Y$.
Defender wants to show $X \sim Y$.

In every round of the game:
Attacker chooses a side (either X or Y) and makes a move (e.g., $X \stackrel{a}{\rightarrow} X'$ or $Y \stackrel{a}{\rightarrow} Y'$). Defender must respond by a move on the other side with the same action (e.g., $Y \stackrel{a}{\rightarrow} Y'$ or $X \stackrel{a}{\rightarrow} X'$). The new configuration is (X', Y').

If one player cannot move then the other player wins. The defender wins every infinite game.

Theorem: $X \sim Y$ iff the defender has a winning strategy.

Differences between Simulation and Bisimulation



$X \simeq Y$, but $X \not\sim Y$.

Weak Simulation and Weak Bisimulation

- Need to abstract from internal actions in system models. E.g., prove the semantic equivalence of a more detailed implementation and an abstract specification.

- Weak simulation and weak bisimulation abstract from internal τ-actions.

- Defined with long moves $\stackrel{\tau^*}{\longrightarrow}\stackrel{a}{\longrightarrow}\stackrel{\tau^*}{\longrightarrow}$, instead of the normal $\stackrel{a}{\longrightarrow}$. Arbitrarily many internal actions before and after every visible action.

On finite-state systems weak equivalences/preorders can be reduced to strong ones by computing the transitive closure w.r.t. τ-actions.

On infinite-state systems this is not effectively possible in general.

Connections to Temporal Logics

- $\sim \subseteq \simeq$ (but not vice-versa).
- On finite labeled transition graphs deciding $\sqsubseteq$, $\simeq$, $\sim$ is polynomial.
- For image-finite graphs $\sim$ is the equivalence induced by branching-time logics. For every branching-time temporal logic L with

  $$\text{Hennessy-Milner Logic} \leq L \leq \text{action-based } \mu\text{-calculus}$$

  (e.g., L = CTL) we have

  $$X \sim Y \quad \text{iff} \quad \text{X,Y cannot be distinguished in L}$$

- Every finite labeled transition graph can be described up-to (strong and weak) bisimulation equivalence in the temporal logic EF (a fragment of CTL) [Jančar, Kučera, Mayr].

General Trends

1. On infinite-state systems, weak equivalences are computationally harder than strong ones.

2. Simulation is computationally harder than bisimulation on almost all 'natural' classes of systems.

Decidability Results

Stronger and stronger positive decidability results for strong bisimulation:

Strong bisimilarity decidable for 1-counter machines [Jančar].

Strong bisimilarity decidable for normed pushdown automata [Stirling].

Strong bisimilarity decidable for general pushdown automata [Sénizergues].

Stronger and stronger negative decidability results for weak bisimulation:

Weak bisimilarity undecidable for general pushdown automata [Srba].

Weak bisimilarity undecidable for normed pushdown automata [Srba].

Weak bisimilarity undecidable for normed 1-counter machines; even for normed 1-counter nets (Petri nets with only 1 unbounded place) [Mayr].

Simulation vs. Bisimulation – Complexity

	Simulation	Bisimulation
normed PA-normed PA	undecidable	decidable
PDA-PDA	undecidable	decidable, EXPTIME-hard
PDA-FS	EXPTIME-complete	PSPACE-complete
BPA-FS	EXPTIME-complete	polynomial

Why is simulation harder than bisimulation ?

One can reduce bisimulation checking to simulation checking (in polynomial time) for any finitely generated class of transition systems that satisfies one of these properties [Kučera, Mayr, 2002].

- The systems can 'test for non-enabledness of actions' (e.g., test for zero). Satisfied, e.g., for pushdown automata.

- The class is closed under parallel composition of processes and synchronization (over the atomic actions). Satisfied, e.g., for Petri nets.

Most classes of systems satisfy one of these requirements. Thus, simulation checking must be at least as hard as bisimulation checking.

Bisim. $\rightarrow$ Sim.: Method 1

Given two labeled transition graphs S and T.

Construct modified transition graphs S' and T' with the property:

$$S \sim T \quad \Leftrightarrow \quad S' \sqsubseteq T'$$

How ? Encode the bisimulation game between S and T in a simulation game between S' and T'.

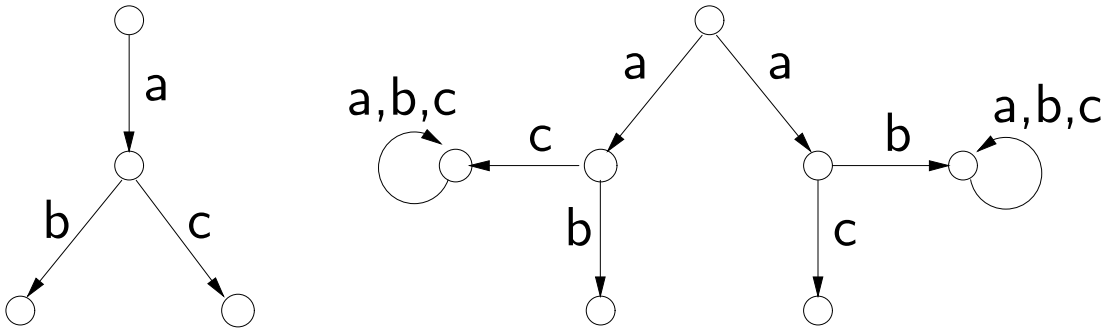
Remark: One can easily reduce simulation preorder to simulation equivalence. Given systems S, T. Define S', T' by $S' \stackrel{a}{\rightarrow} S$, $S' \stackrel{a}{\rightarrow} T$, $T' \stackrel{a}{\rightarrow} T$. Then

$$S' \simeq T' \quad \Leftrightarrow \quad S \sqsubseteq T$$

The Power of the Defender in Simulation Games

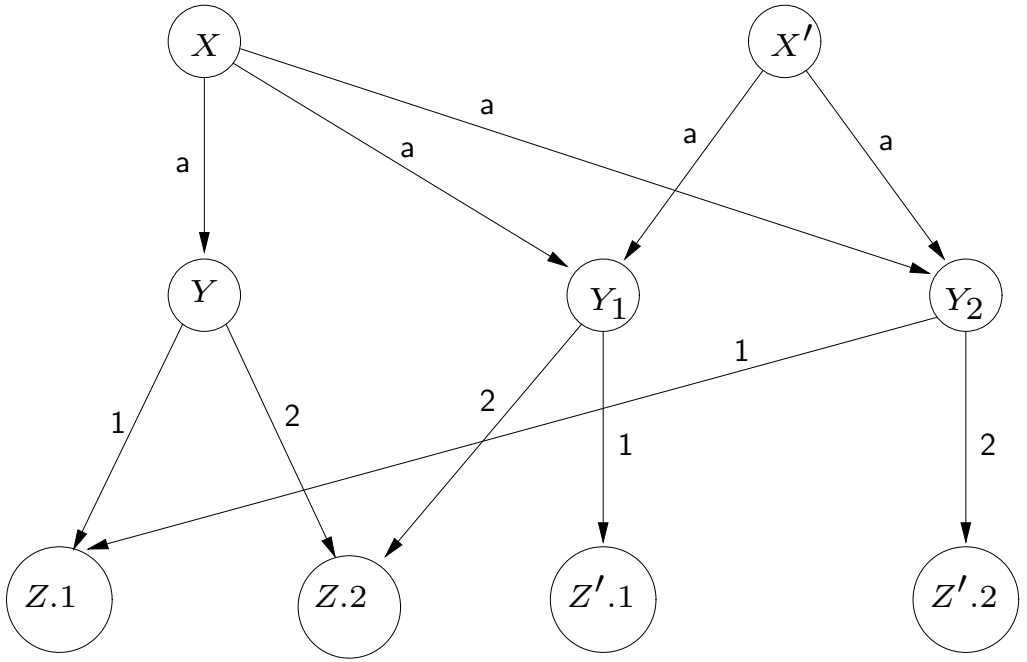
How can the defender in the simulation game force the attacker to something ? He threatens to become universal, i.e, to go to a universal state.

(At a universal state every action is enabled and stays enabled forever, i.e., $\Box\langle Act\rangle true$.)



The Power of the Defender in Bisimulation Games

How can the defender in the bisim. game force the attacker to do something ?
He threatens to make the two processes equal.
A technique by Jančar, Srba. (Here slightly improved):



Bisim. Game $\rightarrow$ Sim. Game

1. Case (simple):

If in the bisimulation game between S and T the attacker moves on the left side, i.e., $S \xrightarrow{a} S_i$, then the defender moves on the right side $T \xrightarrow{a} T_j$.

Directly encoded in the simulation game between S and T.
In simulation games the attacker always moves on the left side.

One round of the bisimulation game encoded in one round of the simulation game.

Bisim. Game $\rightarrow$ Sim. Game (cont.)

2. Case (more complex):

Assume that in the bisimulation game between S and T the attacker moves on the right side, i.e., $T \xrightarrow{a} T_i$ and the defender responds by $S \xrightarrow{a} S_j$.

Encoding in the simulation game:

Attacker moves $S \xrightarrow{a_i} \hat{S}$ (chooses i, demands move to T_i on right side)
Defender responds $T \xrightarrow{a_i} T_{i,j}$ (encodes i, chooses j)
Attacker moves $\hat{S} \xrightarrow{a_j} S_j$ (must make this move; otherwise defender wins)
Defender moves $T_{i,j} \xrightarrow{a_j} T_i$
(All other moves lead to a universal state, $T_{i,j} \xrightarrow{c} U$ for all $c \neq a_j$.)

One round of the bisimulation game encoded in two rounds of the simulation game.

Requirements for the Construction 1.

Problem: The attacker (moving on the left side in S) cannot know which actions are currently enabled on the right side (in T).

What happens if the attacker demands something impossible, i.e., a move in T that is not enabled ?

The defender notices this and goes to a universal state (and thus wins the game). This response must be possible only if the required move is really not enabled.

Analogously, if the defender demands an impossible move on the attacker's side, then the attacker can win by performing a special action.

Conclusion: A test for enabledness of actions is required. Possible, e.g., for pushdown automata, but not for Petri nets.

Formal Def. of Construction 1.

Given two labeled transition graphs S and T (with initial states s_0, t_0).

Define new systems

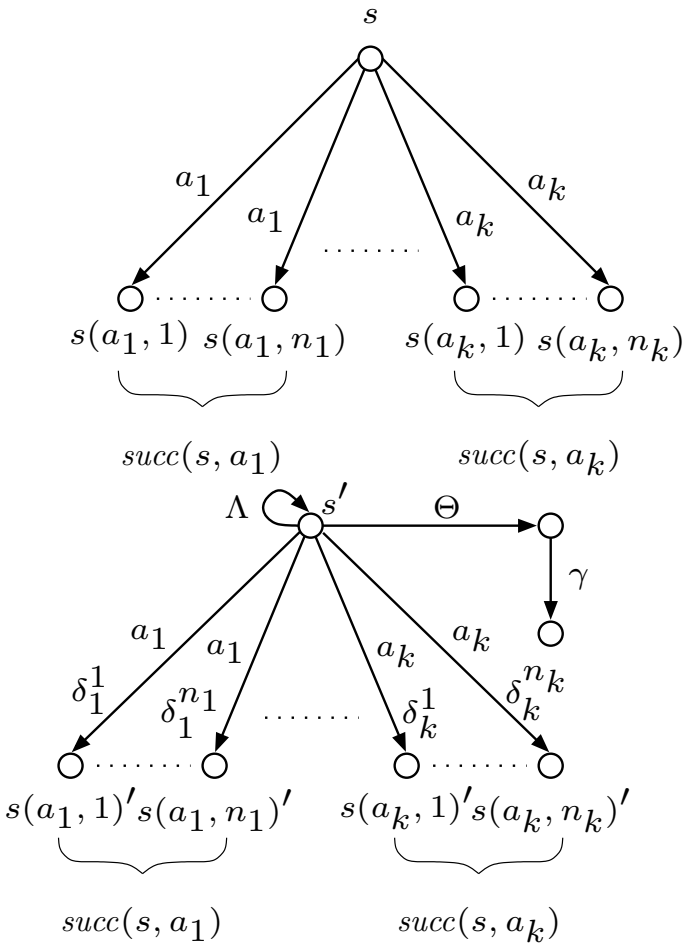
S' := Attacker-extension(S)

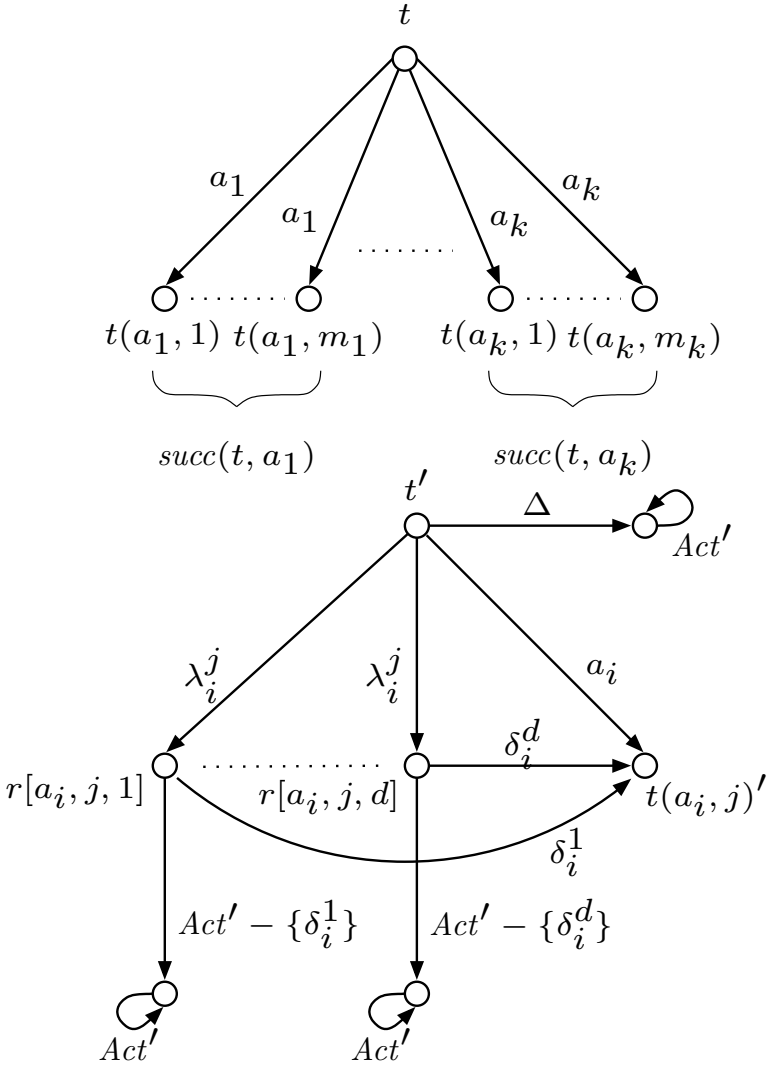
T' := Defender-extension(T)

with initial states s'_0, t'_0.

and obtain

$$S \sim T \quad \Leftrightarrow \quad S' \sqsubseteq T'$$





Bisim. $\rightarrow$ Sim.: Method 2

Given two labeled transition graphs S and T.

Construct new transition graphs $S' := A - comp(S,T)$ and $T' := D - comp(S,T)$ with the property:

$$S \sim T \quad \Leftrightarrow \quad S' \sqsubseteq T'$$

Roughly speaking
$A - comp(S,T) := S \| T$
$D - comp(S,T) :=$ modified-sync.(S,T)

One round in the bisimulation game between S and T is emulated by two rounds in the simulation game between S' and T'.

Intuition: Both S' and T' contain copies of both S and T. They always 'know' which moves are possible in S and T. The construction enforces that these copies are kept consistent.

Bisim. $\rightarrow$ Sim.: Method 2 (cont.)

One round in the bisimulation game between S amd T:

Attacker moves $S \stackrel{a}{\rightarrow} S_i$
Defender replies $T \stackrel{a}{\rightarrow} T_j$

This is emulated in two rounds in the simulation game between S' and T'.

Attacker moves $S' = A - comp(S,T) \stackrel{x_i}{\rightarrow} A - comp(S_i,T)$
Defender replies $T' = D - comp(S,T) \stackrel{x_i}{\rightarrow} D - comp(S_i,\hat{T}_j)$
Attacker moves $A - comp(S_i,T) \stackrel{x_j}{\rightarrow} A - comp(S_i,T_j)$
Defender replies $D - comp(S_i,\hat{T}_j) \stackrel{x_j}{\rightarrow} D - comp(S_i,T_j)$

Conclusion: No test for enabledness of actions required, but parallel comp. and sync. are needed. Works, e.g., for Petri nets and subclasses, but not for PDA.

Formally: Method 2

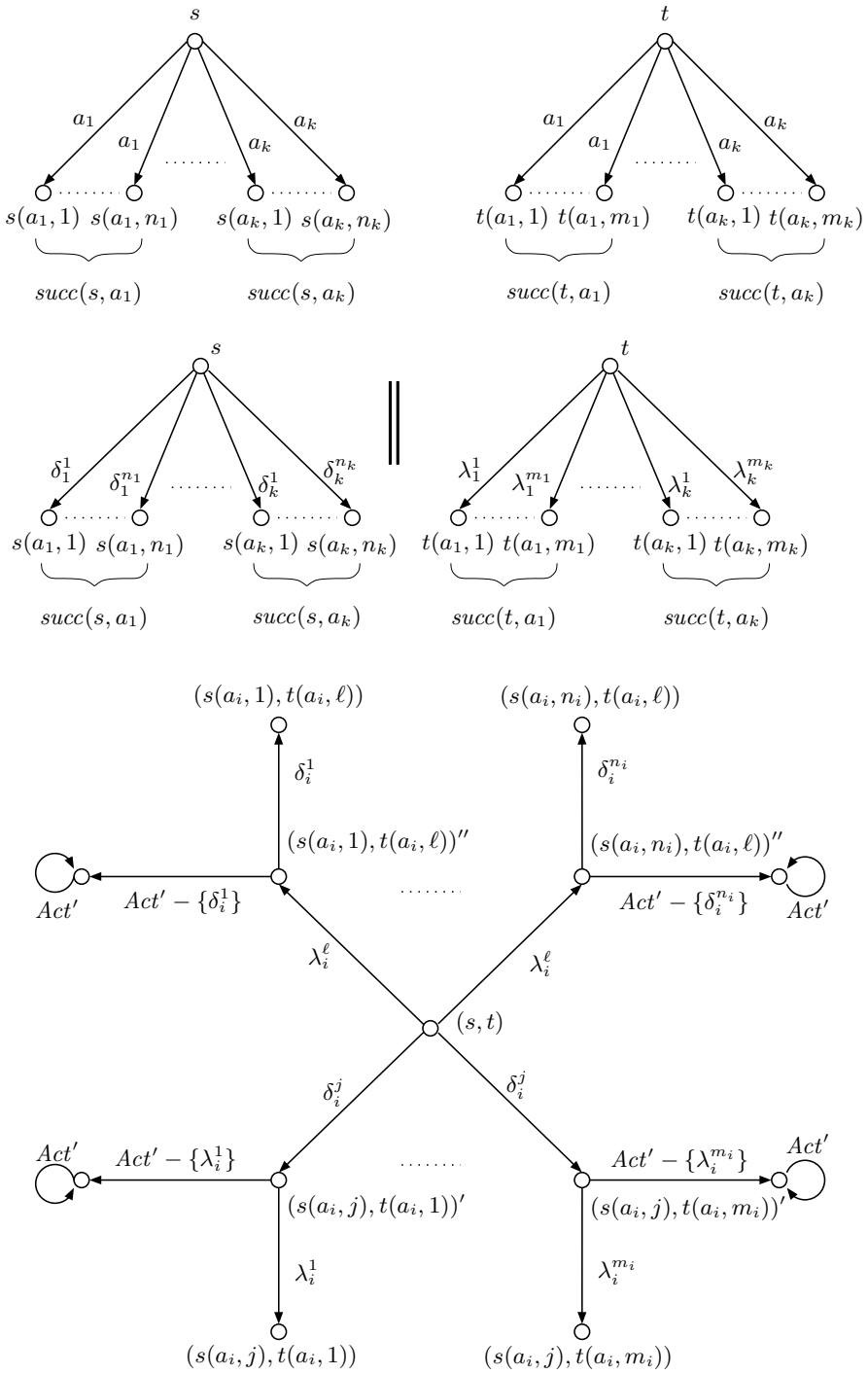


Fig. 2. States s and t of $\mathcal{S}$ and $\mathcal{T}$, then the A-composition of $\mathcal{S}$ and $\mathcal{T}$, and finally the D-composition of $\mathcal{S}$ and $\mathcal{T}$.

Applied Verification

- Development of the practical tool IBOC (Incomplete Boundedness Checker). Analyzes systems described in UML RT and PROMELA.

- Solves questions for systems with asynchronous communication via buffers.
  - Can one get a buffer overflow?
  - Are the reachable buffer lengths bounded?
  - If yes, then by what constant?

- Automatic incomplete boundedness check for communication buffers between capsules/processes, without exploring the state space.
  Gives answers 'YES' or 'NOT PROVEN $\rightarrow$ hints for abstraction refinement'.

- Works by static analysis, without exploring the reachable state-space.

- Can derive upper bounds on reachable communication buffer content lengths in most cases.

Applied Verification (cont.)

- Technique 1: Abstract interpretation and overapproximation.
  Abstraction preserves
  - Simulation preorder.
  - Upper bounds on reachable channel contents.
- Technique 2:
  - Static analysis for tracking ranges of crucial variables.
  - Analyzing possible combinations of the effects of cycles in the control-flow graph. $\longrightarrow$ Linear programming.
- Scalable: Works in polynomial time on average-case UML RT/PROMELA models. (Exponential in the worst case).
- Approach can be generalized to related problems, e.g., memory allocation (memory leaks!), dynamic process creation/destruction, etc.

Case study: UML RT

- PBX 'Private Branch Exchange', a telephone switching system.
- System defined in UML RT
  - 29 capsules
  - 736 control states
  - 57 communication buffers
  - 308 different message types
- $\sim 10^{40}$ control-state combinations alone, not even counting buffer contents.

Run times of the IBOC-tool (on 1 GHz Pentium III):
30 seconds to prove boundedness of the whole system.
1 minute to compute upper bounds on the lengths of all 57 communication buffers.

$\longrightarrow$ Paper to appear in TACAS 2004.

Case Study: PROMELA

- CORBA General Inter-ORB Protocol (GIOP).
- System model defined in PROMELA.
  - 5 processes
  - 135 control states
  - 11 communication channels
  - 108 different message types
- IBOC proved that there are only two causes of unboundedness:
  - Flooding by user requests.
  - Unboundedly many register messages for object migrations.
- If these two causes are removed then IBOC proves boundedness and computes upper bounds in about 10 seconds (on 1 GHz Pentium III).

$\longrightarrow$ Paper to appear in SPIN 2004.

Conclusion

- Automated verification of infinite-state systems is harder than for finite-state ones, but feasible.

- It requires careful analysis of the problem complexities in all input parameters, and the combination of many techniques, e.g.,
  - Abstraction
  - Semantic equivalences
  - State-space reduction techniques
  - Symbolic representation of infinite sets and relations
  - Over- and under-approximation
  - Acceleration

