Symbolic Model Checking and Simulation with Temporal Assertions

University of Tübingen

Computer Engineering Department

Roland J. Weiss

FDL'04 – Lille, France

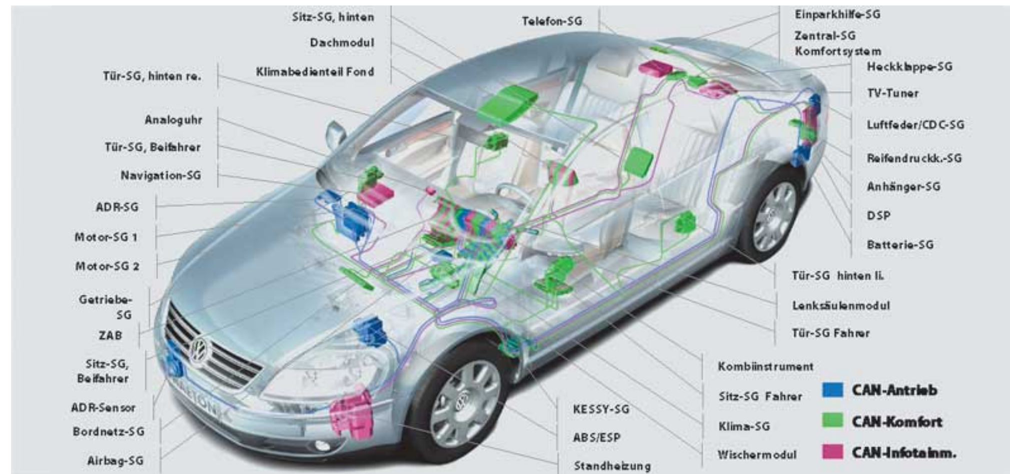
September 16th 2004

Agenda

- Introduction to verification
  - Motivation
  - Classification
- Property specification
  - Property patterns
  - Property specification formalisms
- Property checking
  - Simulation and formal properties
  - BDD-based formal verification techniques
  - Semiformal approach
- Conclusions

Motivation

- Study by Center of Automotive Research (CAR) for ADAC
  - 59.2% of car breakdowns caused by faults in automobile electronics & electricity
  - 50.5% 5 years ago, overall figures constant
  - All car brands are affected



Motivation

- Complexity of modern HW/SW-systems poses new challenges
  - Meet requirements
  - Response times (real-time constraints, DOS)
  - Minimize downtime
  - Repeatable, cost-effective development
  - Maintenance

Verification Goals

- Gain confidence in HW/SW-systems
  - Comply to specification
  - Trust performed changes

- Improve design flow
  - Early error detection
  - Monitor deployed systems
  - Make complex systems tractable

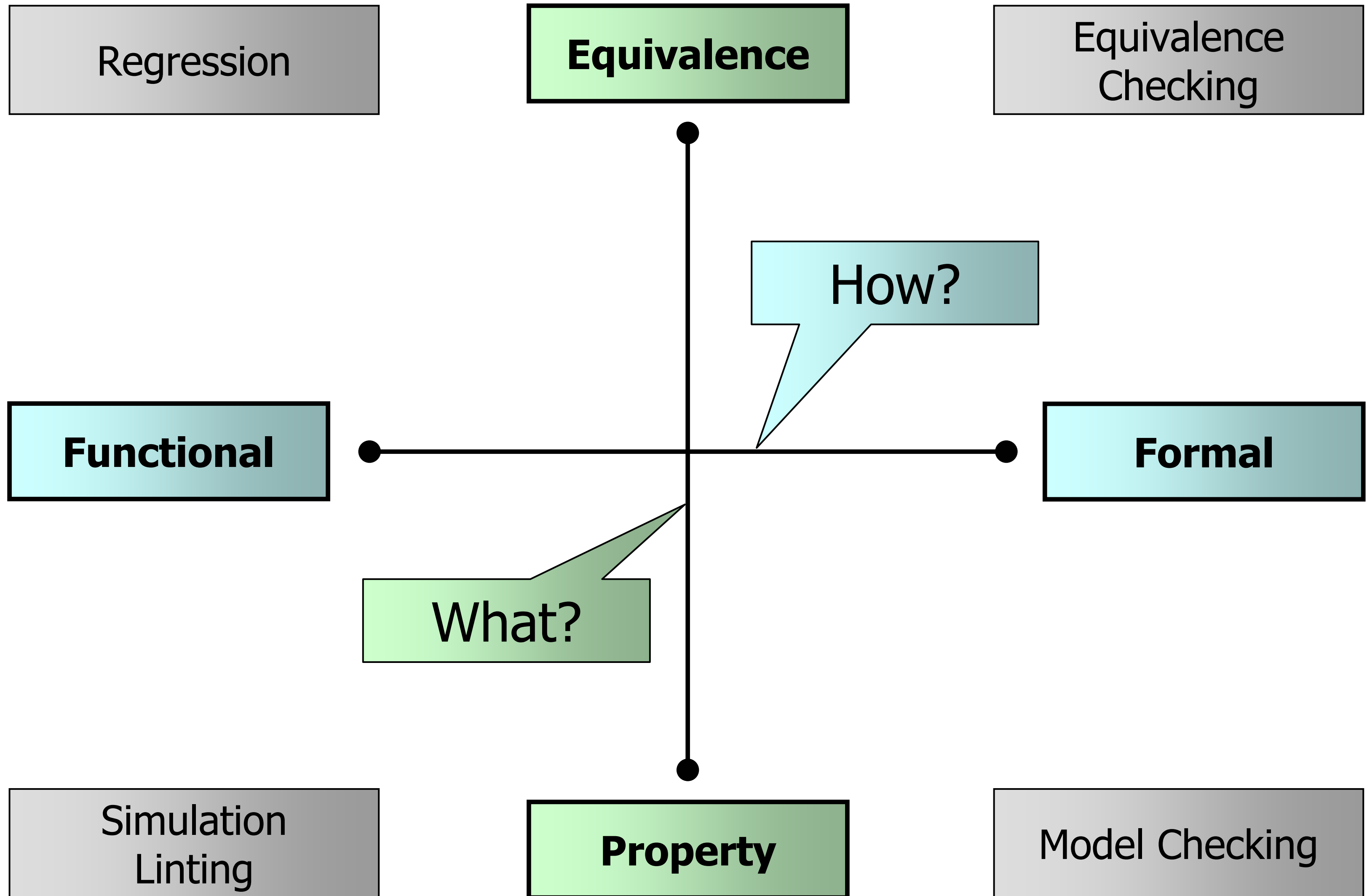
Driving Forces

1. Safety-critical systems
   - Error-free functioning required: Malfunctioning endangers human life or environment
   - Transportation industries, medical systems, power plants, …

Driving Forces

2. Economical risks
   - Late error detection more expensive
   - Erroneous chips produce deficits
   - Ariane 5 crash
   - Pentium floating point unit bug

Verification Classification



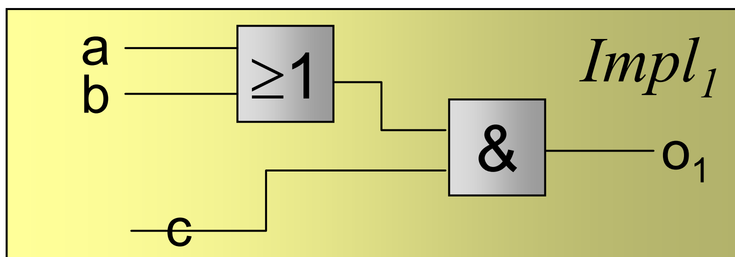
Functional vs. Formal Verification

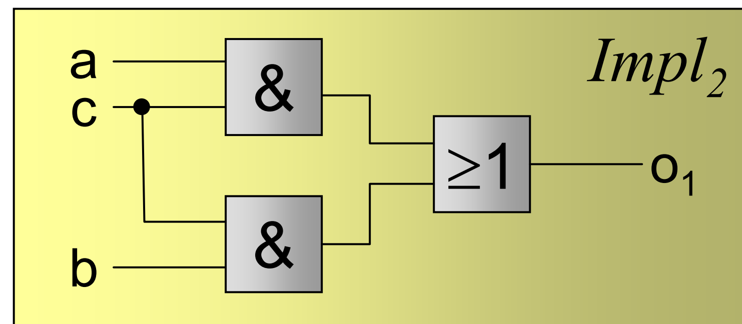
	Formal	Functional
Coverage	100%	incomplete
Model size	small – medium	large

- Formal verification tools provide 100% coverage only for present properties
- Functional verification tools provide nonexhaustive coverage but handle large designs

Equivalence Checking

- Given:
  - HW implementation $Impl_1$
  - HW implementation $Impl_2$
- Verification task:
  - Prove that $Impl_1$ is equivalent to $Impl_2$
  - Notation: $Impl_1 = Impl_2$





Property Checking

- Given:
  - HW implementation $Impl$
  - Specification $Spec$ of design intent
- Verification task:
  - Prove that $Impl$ meets $Spec$
  - Notation: $Impl \rightarrow Spec$ or $Impl \models Spec$
- Remainder of talk focuses on Prop. Checking

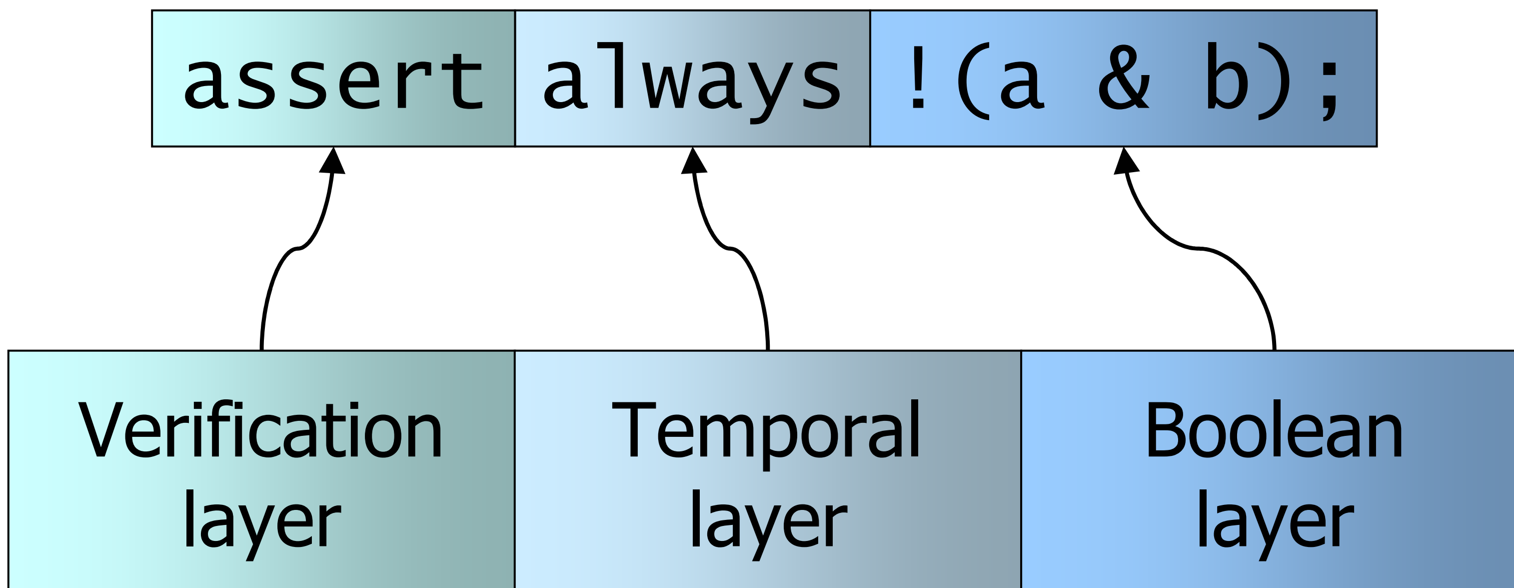
Property Specification

What is a Property?

- A *property* is a description of design intent. [Coelho and Foster, 2004]

- Properties are composed of three layers
  1. Boolean layer: Propositions and Boolean connectives
  2. Temporal layer: Operators for temporal reasoning
  3. Verification layer: Indicators for verification tools how to apply the property

Layered View of Properties



- Sometimes also modeling layer:
  - Means to model behavior of design inputs
  - PSL/Verilog: integer ranges, structures, non-determinism, built-in functions

Property Categories

- **Safety:** "something bad does not happen"
  - Any path violating the property has finite prefix for which every extension violates the property
- **Liveness:** "something good eventually happens"
  - Any finite path can be extended to a path satisfying the property
- **Fairness:** "something will occur infinitely often"
  - Any infinite path will contain infinitely fair states

Property Specification Patterns

- Goal
  - Present common properties in reusable form
  - Capture knowledge of specification experts
  - Further practical usage of formal verification tools
  - Repeat success of design patterns

- Origin
  - Matthew Dwyer (Kansas State University) et. al.
  - Homepage: http://patterns.projects.cis.ksu.edu
  - Original papers [Dwyer et. al., 1998 & 1999]

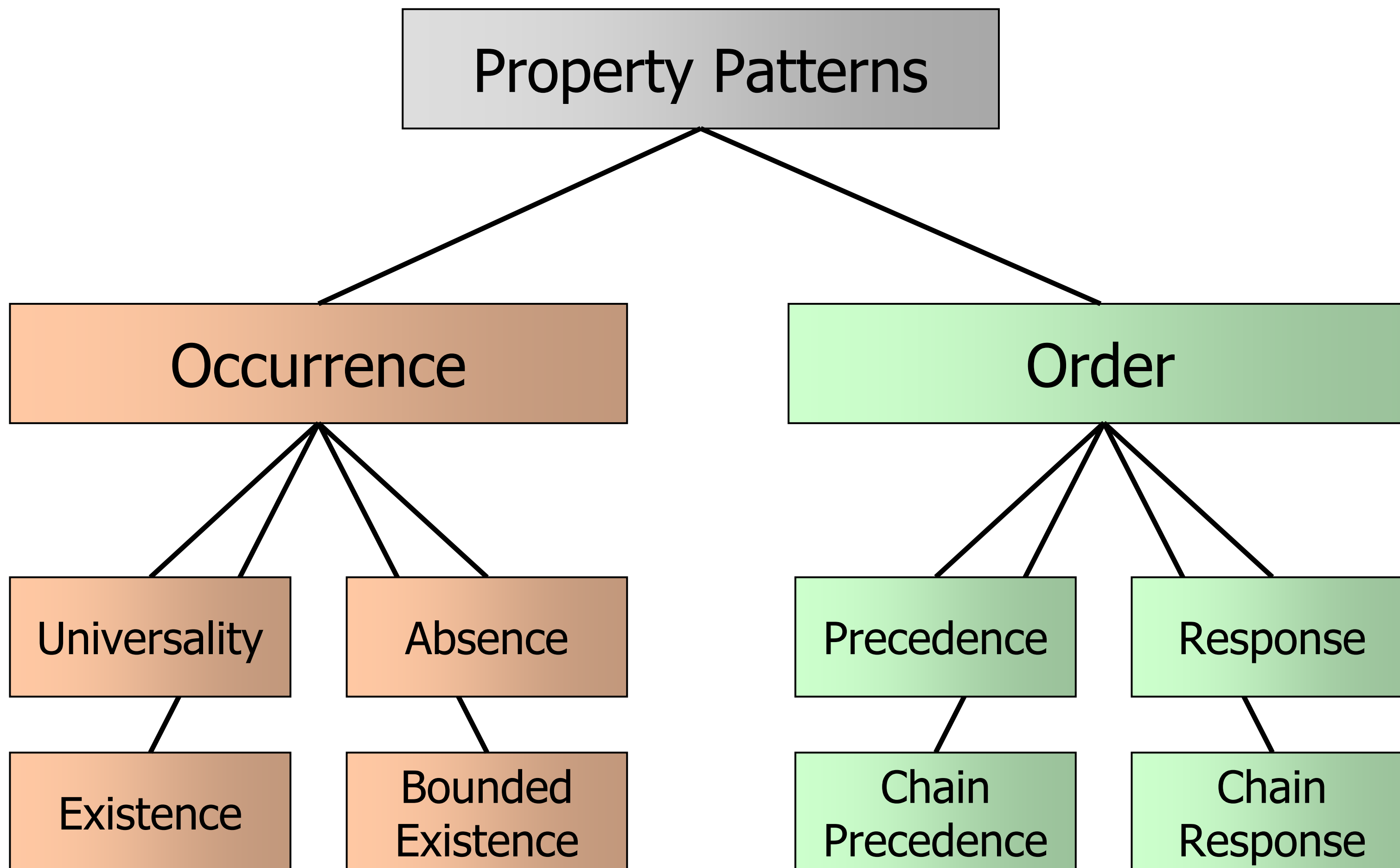
Constituents of Specification Patterns

- The pattern’s name/names
  - Precedence (Enables)
- Statement of the pattern’s intent
  - Pair of events/states *S* & *P*
  - Occurrence of *S* precondition for occurrence of *P*
- Mappings into specification formalisms
  - CTL, LTL
  - QRE (Quantified Regular Expressions)
  - GIL (Graphical Interval Logic)
  - INCA

Constituents of Specification Patterns

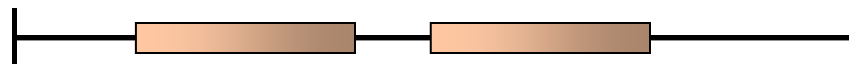
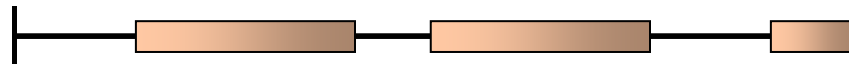
- Examples of known uses
  - Concurrent systems: resource (lock) granted in response to a request
- Relationships to other patterns
  - Chain precedence pattern allows multiple separate states/events to constitute *S* and *P*

Pattern Hierarchy



Pattern Scope

- Every pattern codification is given for 5 scopes (scope → extent of program execution for which property must hold)
  - Global
  - Before *Q*
  - After *Q*
  - Between *Q* and *R*
  - After *Q* until *R*

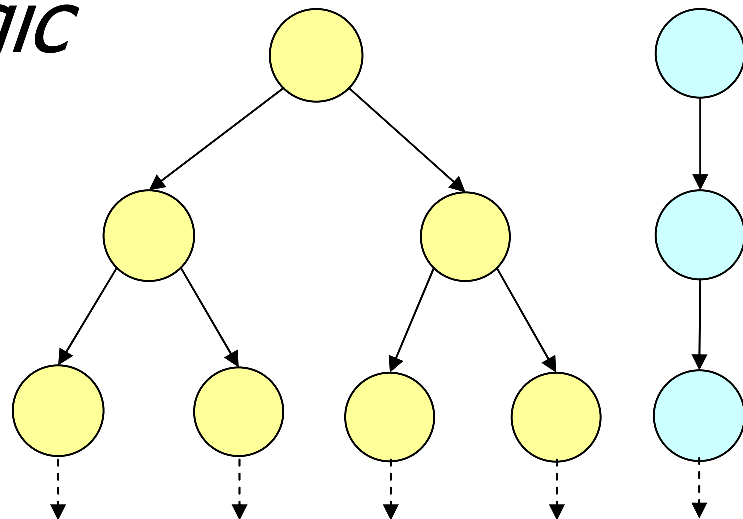



Survey of Pattern System

- 555 specifications, > 35 sources
  - 92% of specs matched pattern systems (8 base patterns with intended variations)
  - 80% of specs are Response (245), Universality (119) and Absence (85) instantiations
  - 80% of specs use global scope

Specification Formalisms

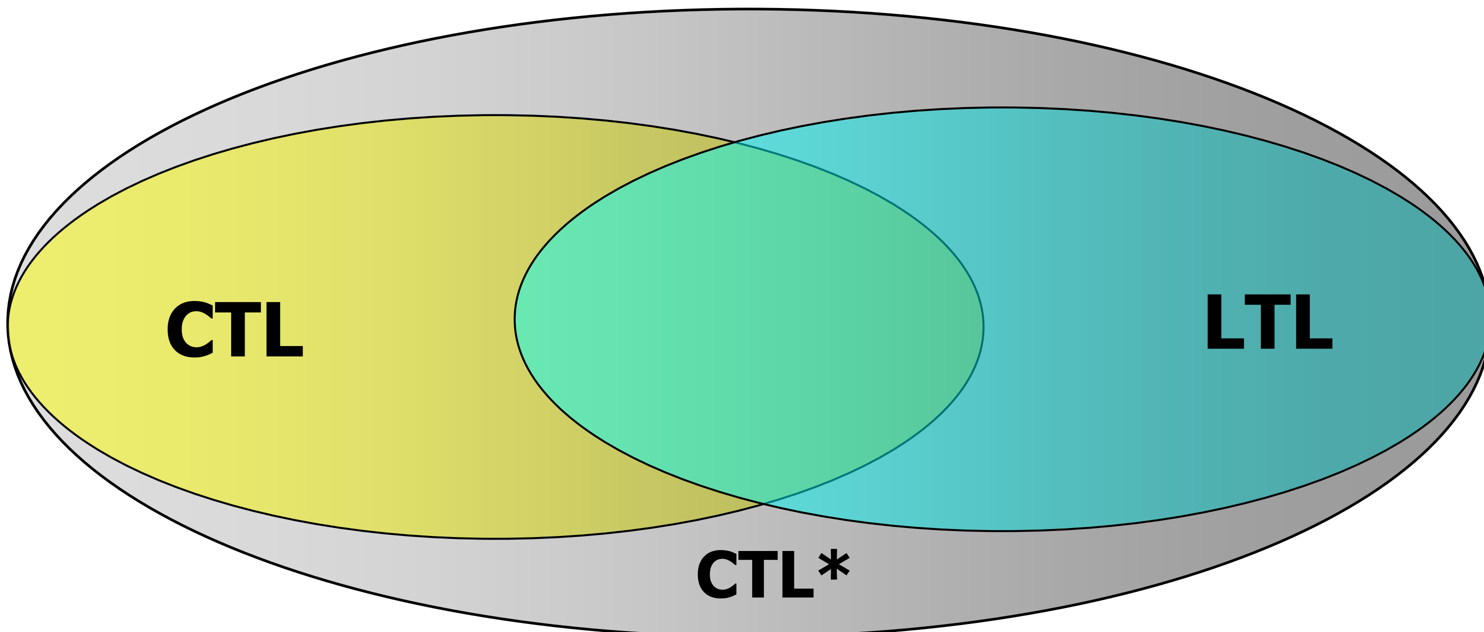
- Formal verification tools typically use flavors of *(propositional) temporal logic*



- Criteria of time model:
  - Linear vs. branching time
  - Discrete vs. continuous time
  - Time points vs. time intervals
  - Past time / future only reasoning

CTL*, CTL and LTL

- Propositional logics with temporal operators
  - CTL* subsumes CTL and LTL
  - LTL (Linear Temporal Logic) – linear time
  - CTL (Computation Tree Logic) – branching time



CTL*, CTL and LTL

- Operators for linear time (temporal operators)
  - **G** (**g**lobally), **F** (**f**inally/eventually), **X** (ne**x**t), **U** (**u**ntil)
- Operators for branching time (path quantors)
  - **A** (on **a**ll paths), **E** (there **e**xists a path)

- LTL: only linear time operators
- CTL: linear & branching operators, but every path quantor must be followed by temporal operator

- Semantics defined in terms of paths

Example and Applications

- Absence Pattern
- Mutual exclusion of signals a, b
  - LTL: $\mathbf{G}(\neg(a \wedge b))$
  - CTL: $\mathbf{AG}(\neg(a \wedge b))$
- LTL considered more intuitive, natural in simulation contexts (traces)
- CTL well suited for model checking algorithms
- Branching vs. linear time comparison: [Moshe Vardi, 2001]

Time Bounds

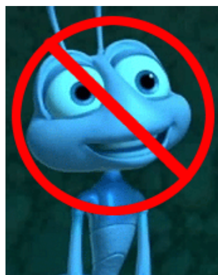
- Extensions with time bounded operators exists both for CTL and LTL

- FLTL [Ruf et al., Date 2001]
  - **X**[n]φ → φ holds in n steps
  - **F**[m,n]φ → φ eventually becomes true in m…n steps
  - **G**[n]φ → φ holds for next n steps
  - φ**U**[m,n]ϕ → φ holds until ϕ becomes true within next m…n steps

Benefits of Properties

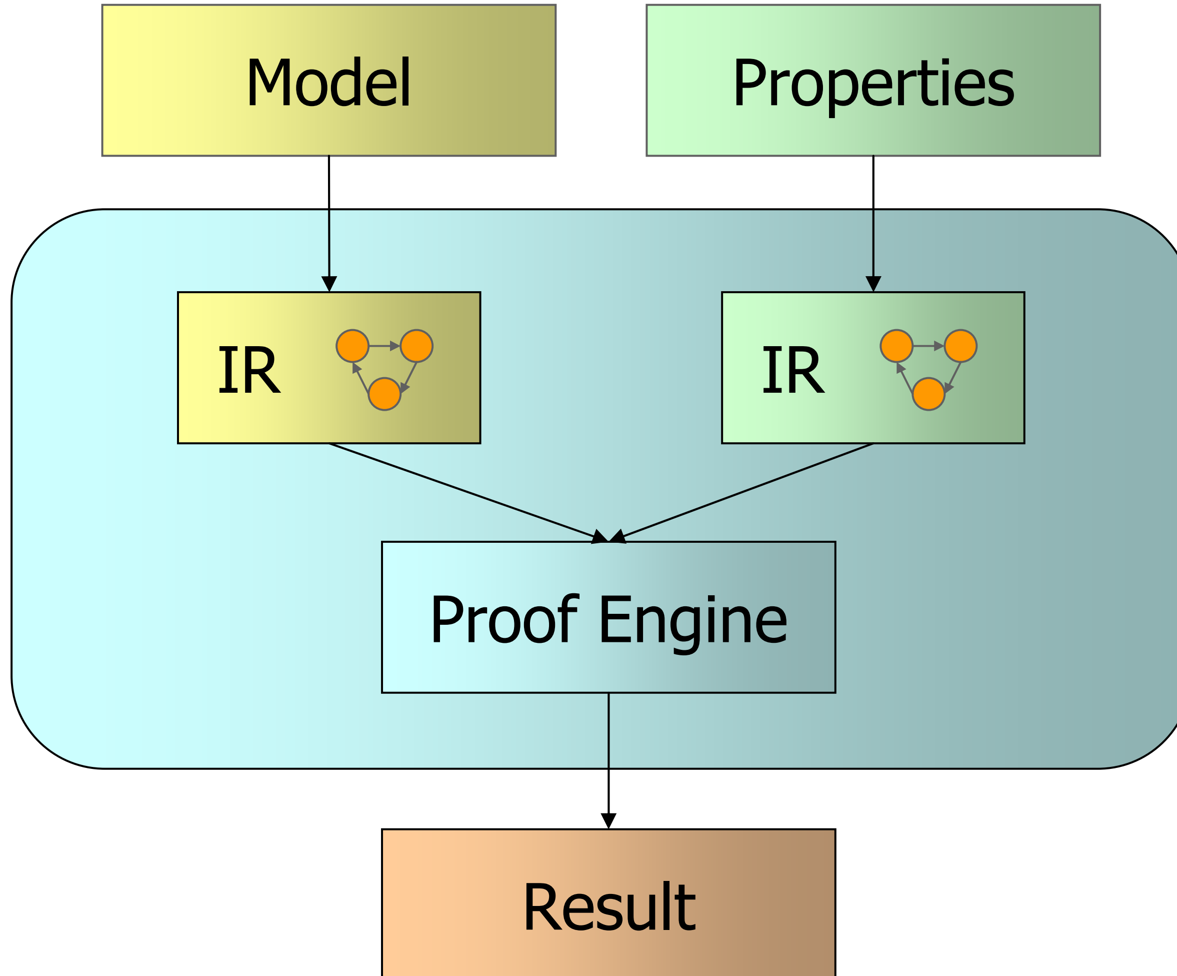
- Improved understanding of system and its requirements

- Improved communication of design intent among involved parties

- Properties indicate problems close to error source 

Property Checking

General Approach

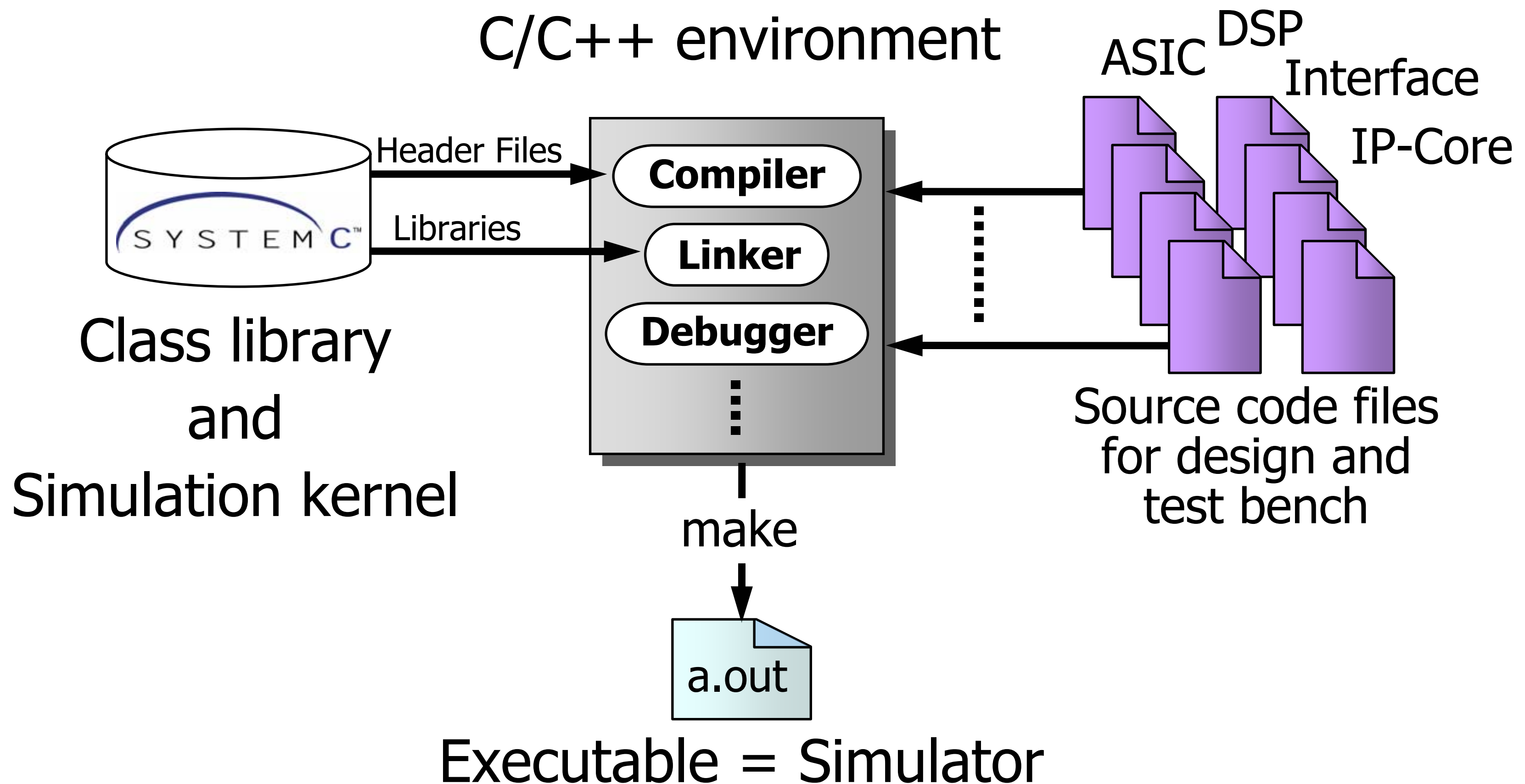


Today

1. Property Checking in SystemC
   - Model in SystemC
   - Properties in PSL (and FLTL…)
   - Functional simulation with formal properties

2. Property Checking in SymC
   - Model as transition graph (smv dialect)
   - Properties in PSL / FLTL
   - Symbolic simulation

What is SystemC?



What is SystemC?

- C++ library for modeling HW/SW-systems
  - Processes: simultaneously executing hardware units
  - Modules: structural and hierarchical composition
  - Channels, ports, interfaces: communication of processes
  - HW specific data types: signals, bitvectors, …
  - Events (e.g. clock edges)
- Simulation kernel
  - Event-driven simulation with nonpreemptive processes
  - Notion of time with clock objects

➔ Domain specific language with simulator

Property Checking in SystemC

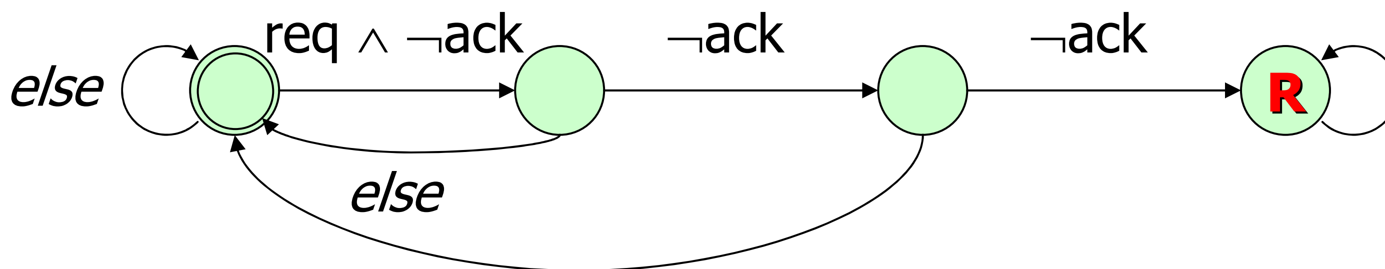
- Model system with SystemC
  - Standard modeling process

- Augment model with property specifications
  - Property translation to internal representation
  - Static vs. dynamic property definition
  - Integration into simulation engine

Property Representation – AR-Automata

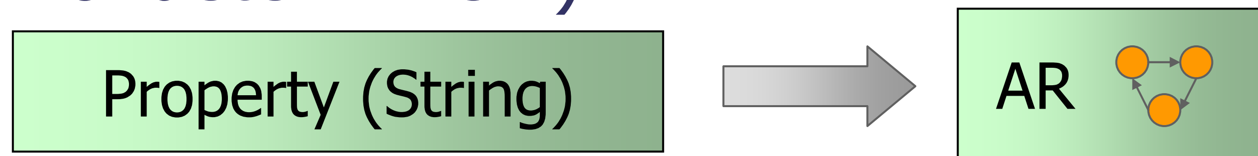
- Deterministic Finite State Machine (FSM)
  - Well suited for reasoning about finite traces
  - Three-valued logic: Accept, Reject, Pending
  - Signals of the model are FSM's input

- Example: $\mathbf{G}(req \rightarrow \mathbf{F}[2]\ ack)$



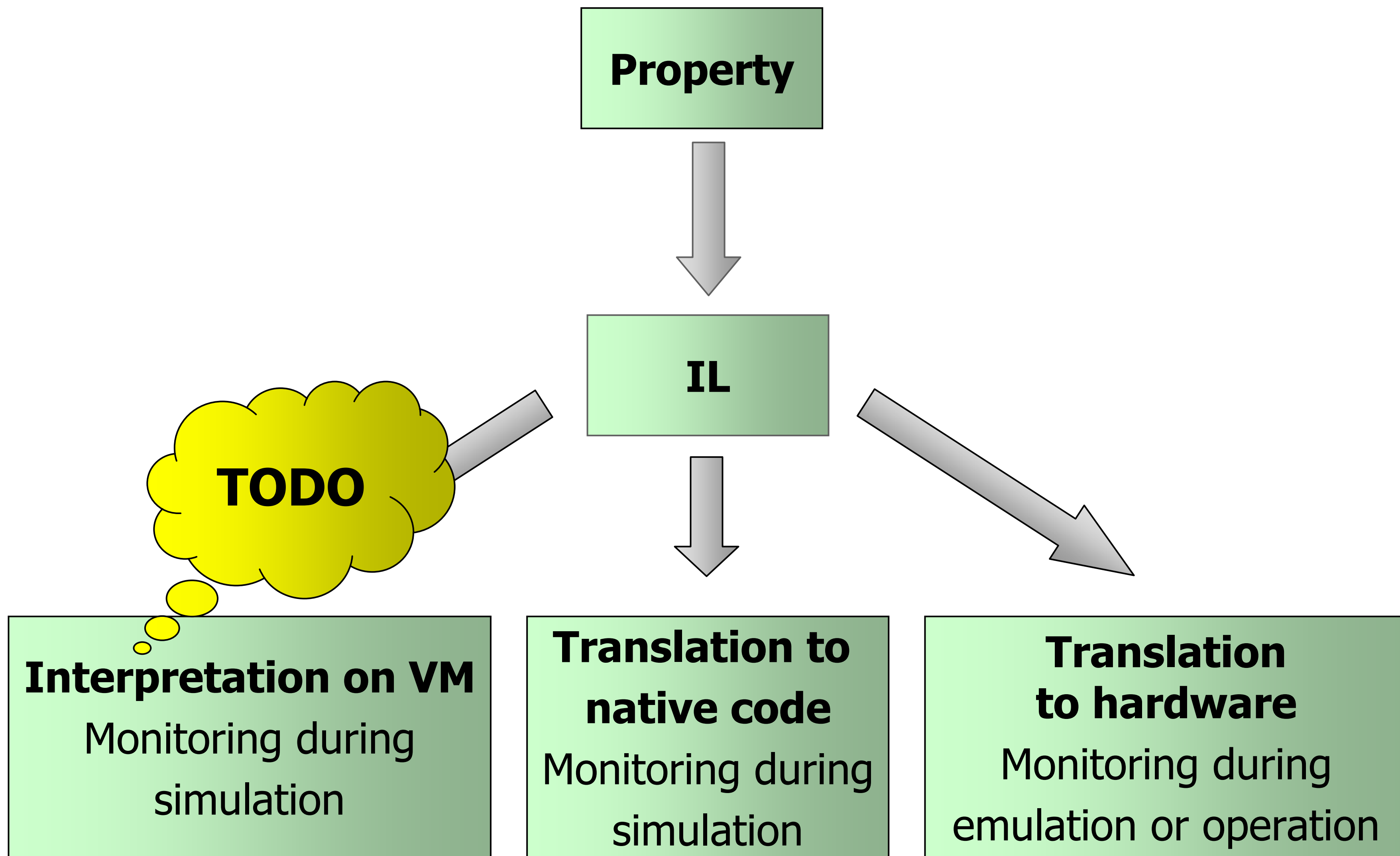
AR-Automata Construction

- V1 – Java implementation
  - Bottom-up composition starting at propositions
  - Applies standard FSM algorithms (minimization, removing nondeterminism)



- V2 – C++ implementation
  - Conversion to space-efficient IL code
  - FSM algorithms have to be adapted to IL
  - IL code can be translated to FSM instance or interpreted

IL - Overview



IL – Intermediate Language for Properties

- Executable, linear representation of properties

Category	Statement	Semantics
Time	`WAIT n`	Wait n steps
Compare	`CHK s`	Check signal s

Jump	`JMP n`
	`JEQ n`
	`JNE n`
		Jump to address n

Return	`RNE T/F`
	`RET T/F`
	`REQ T/F`
		Terminate checking with true / false result

- Bottom-up construction with merge operation (IL code between `WAIT` statements)

IL - example

- $\mathbf{G}(req \rightarrow \mathbf{X}\ ack)$

```
00000000:00000001 CHK  00000000     // req → 0
00000004:0000002a JEQ  0000002c     // ack → 1
00000008:0000005f WAIT 1,5
0000000c:00000001 CHK  00000000
00000010:00000012 JEQ  00000020
00000014:00000011 CHK  00000010
00000018:fffffff0 JNE  00000008
0000001c:1fffffff RET  0
00000020:00000011 CHK  00000010
00000024:00000008 JNE  0000002c
00000028:1fffffff RET  0
0000002c:0000005f WAIT 1,5
00000030:ffffffd0 JMP  00000000
```

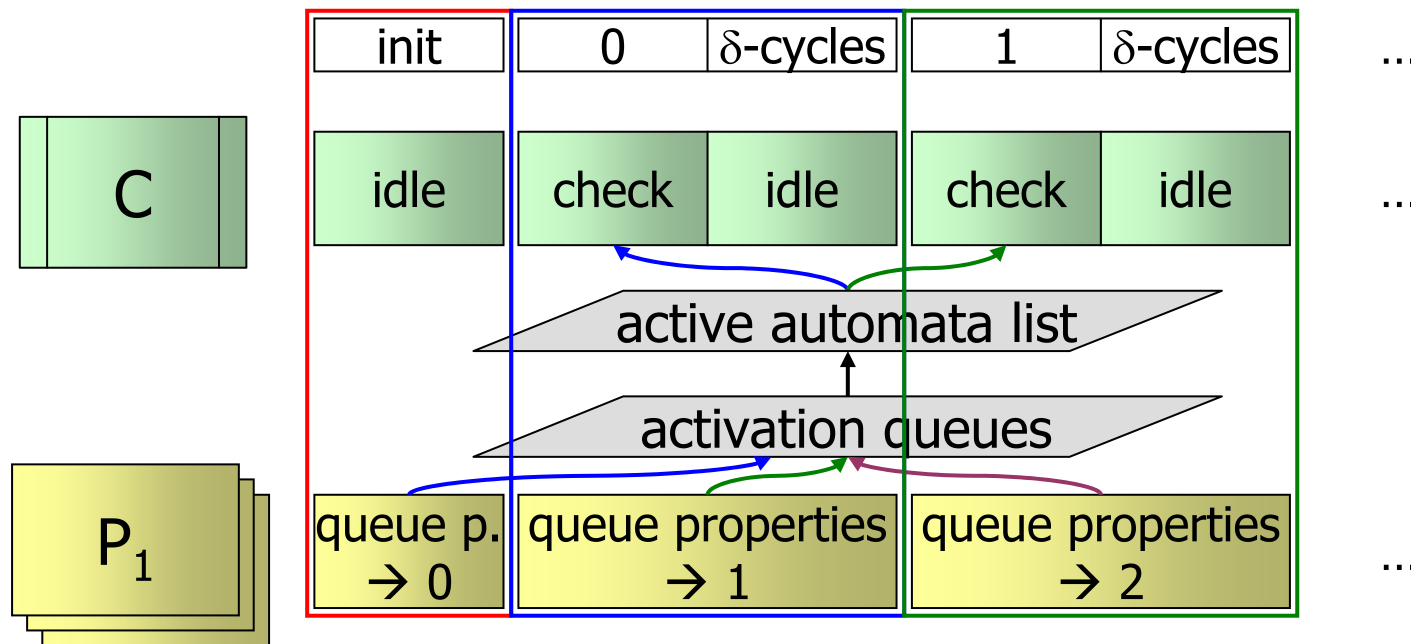
Checker Interface

```
#include “sc_check.h”
…
Module::method()
{
   …
   sc_check(“G[10] !( a & b )”); // Add prop.
   …
}
```

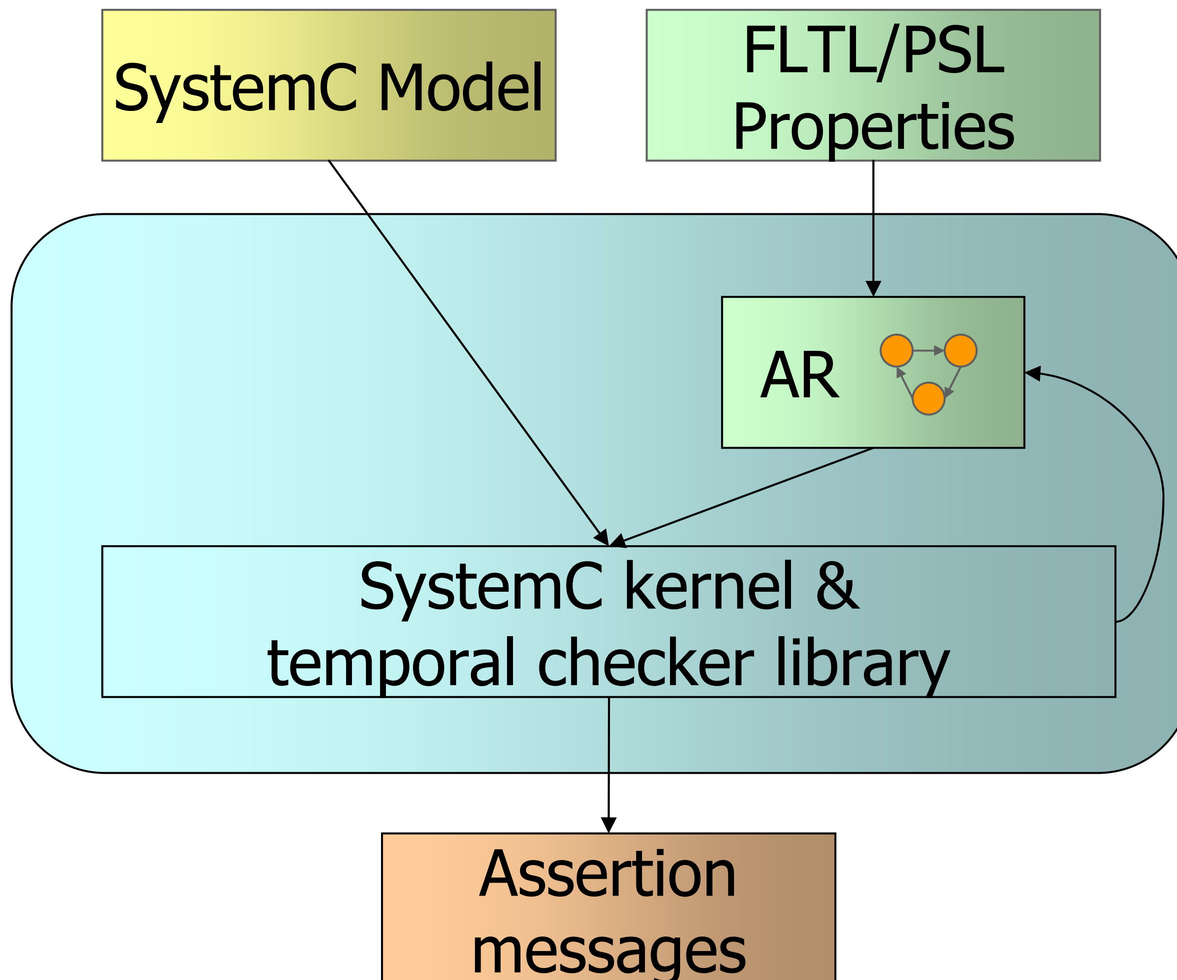
- Properties can be added dynamically (during simulation) → checking starts in next cycle
- Specification part of model code
- Different flavors (check, report, monitor)

Integration into Simulation Engine

- Dedicated process for executing AR-automata in parallel to system model processes
- Check signal values before δ-cycles



SystemC Property Checking



Open Issues

- Optimizations for IL
  - Hierarchical property cache
  - Linked IL code (avoid state explosion)
- SystemC 2.01 integration cumbersome
  - No callbacks into simulation phases
  - Maybe V2.1?
- Currently limited to one checker clock
- Extend checker from signal level checking to transaction level checking (already possible in limited way)

Formal Property Checking: Model Checking

- Check properties on all execution paths
  - Prove properties to always hold or provide counter example
  - Full coverage, as long as set of properties sufficient…
- Major technologies: BDDs (Binary Decision Diagrams) & SAT solvers
  - Recent work combines both approaches: [Cabodi el al.]

Basic approach

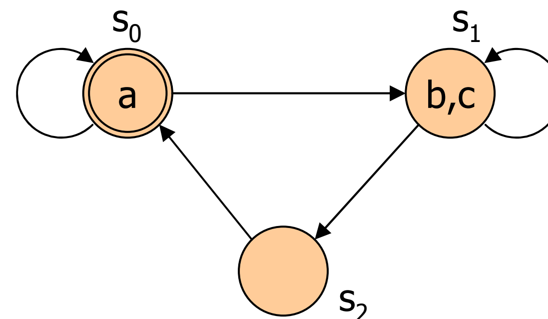
- Represent system model as Kripke structure $\mathcal{M}=(\mathcal{S}, \mathcal{R}, \mathcal{L})$ (or some derivative)
- Represent system specification as temporal logic formula f
- Find set of all states in S that satisfy f:

$$\mathcal{A} = \{s \in \mathcal{S} \mid \mathcal{M}, s \models f\}$$

→ System satisfies spec if all initial states in $\mathcal{A}$!

Kripke Structures

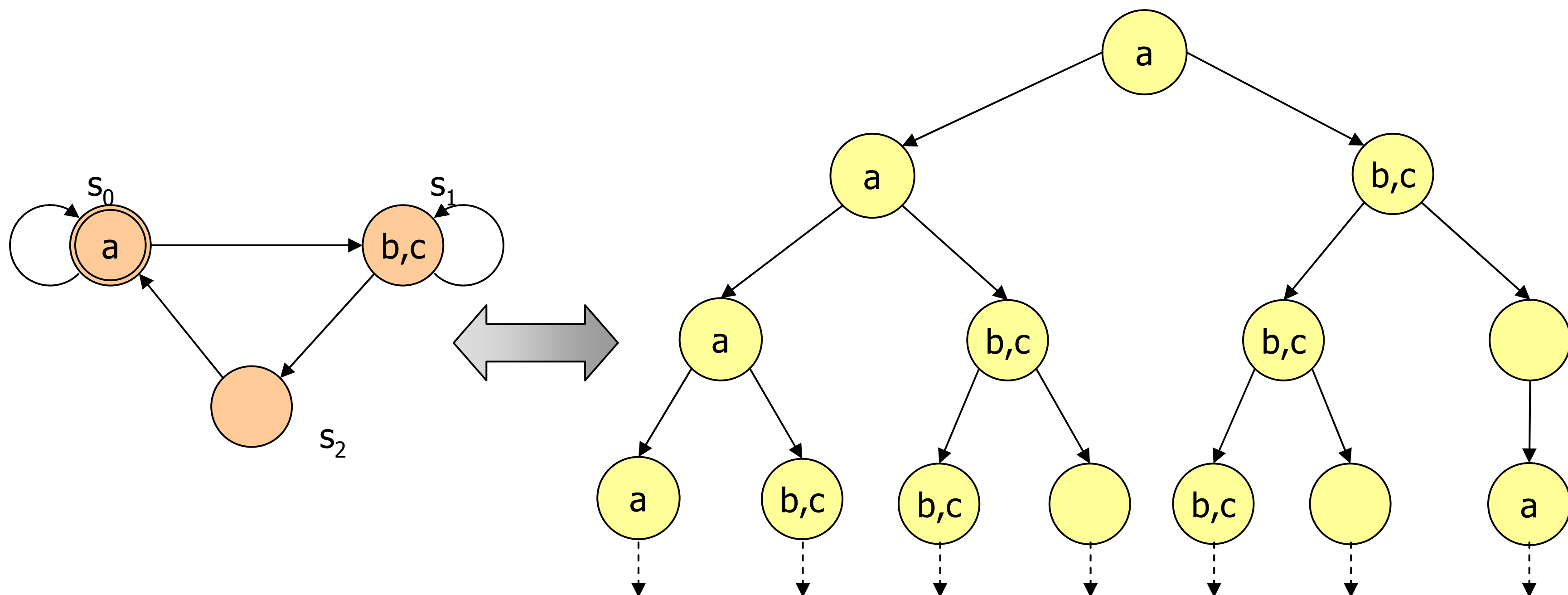
- CTL semantics is defined over Kripke structures
- Used to represent system models



- Kripke structures consist of
  - Set of states $\mathcal{S} = \{s_0, s_1, s_2\}$
  - Initial state set $\mathcal{S}_0 = \{s_0\}$
  - Transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$
  - Set of atomic propositions $\mathcal{P} = \{a, b, c\}$
  - State labeling function $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}}$

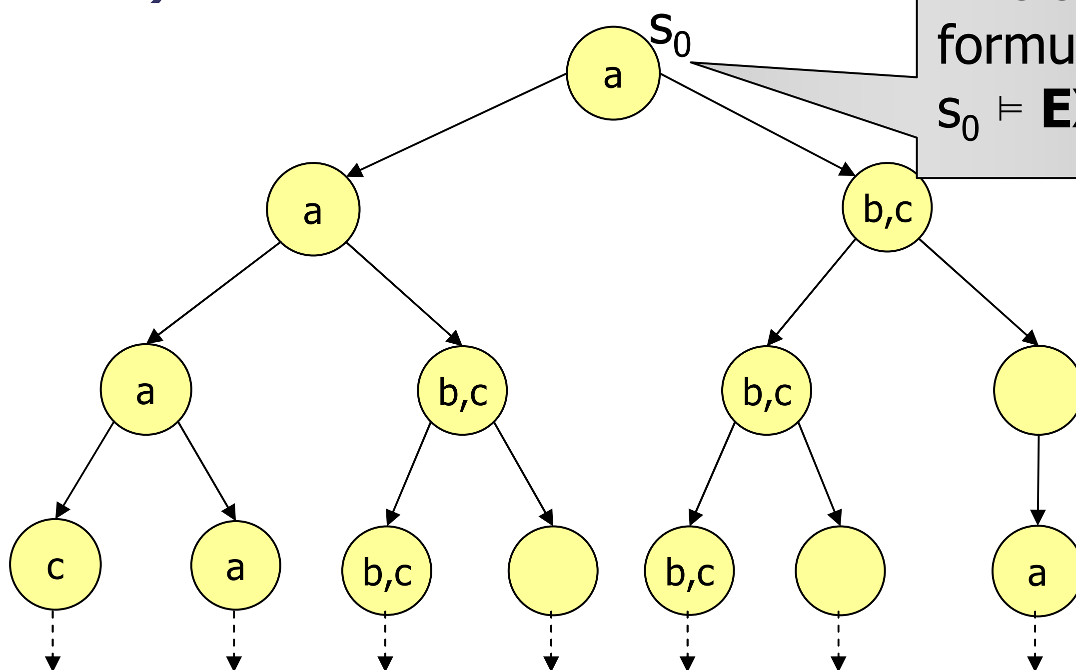
Computation Tree

- Obtained by *unrolling* Kripke structures
- Used to define CTL Semantics
- Path view to Kripke structures



Examples

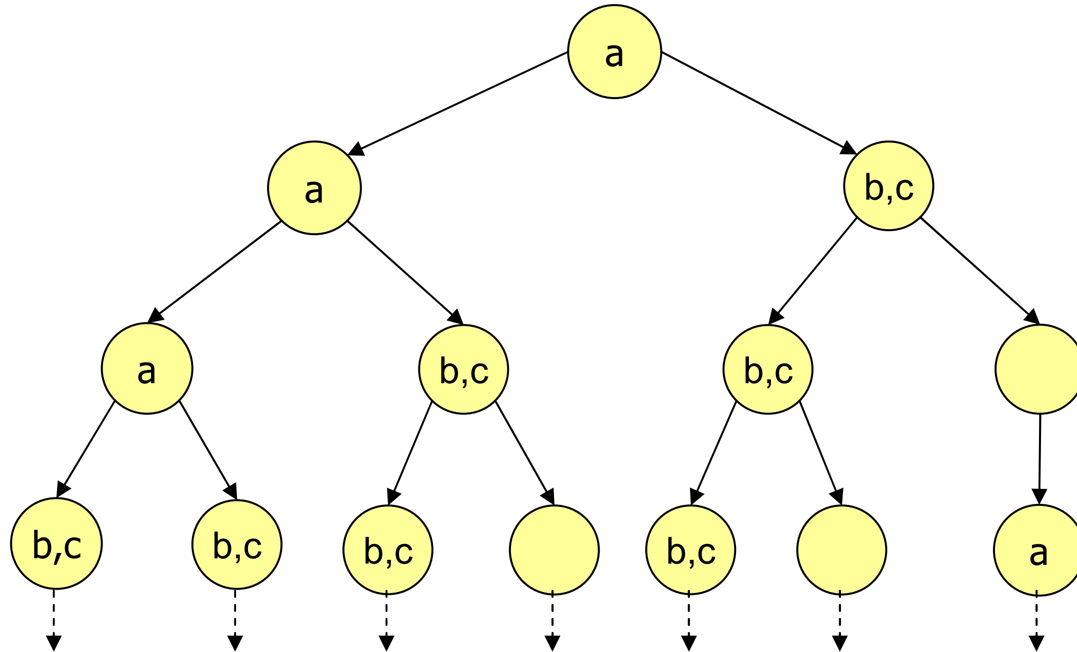
- **EX**(b∧c)
  - There exists a path such that in the next cycle (b ∧ c) is true



This state satisfies the formula. We write: $s_0 \models$ **EX**(b∧c)

Examples

- **AF**(b∧c)
  - on **all** paths, a and b become **eventually** true



Explicit & Symbolic Model Checking

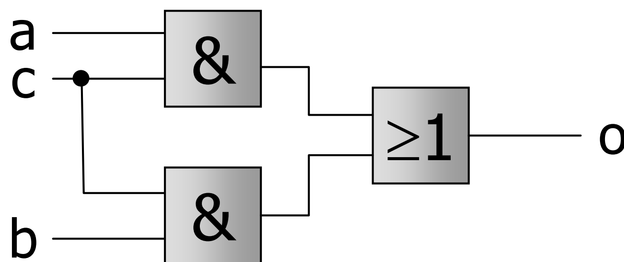
- First algorithms operated on explicit representation of Kripke structures
  - Labeled, directed graph
  - Nodes represent states $\mathcal{S}$
  - Arcs represent transition relation $\mathcal{R}$
  - Labels on nodes represent labeling $\mathcal{L}$

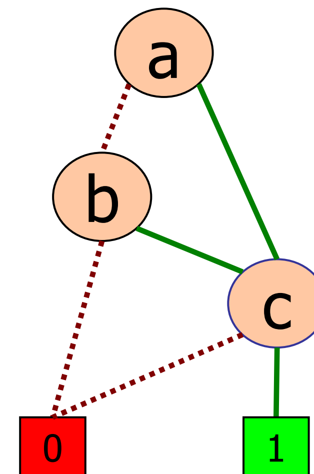
- Algorithms do not scale to large systems: **State explosion problem**

➔ Use efficient symbolic representation

Symbolic Representation with BDDs

- ROBDDs
  - Canonical representation for Boolean functions
  - Support efficient function manipulation (APPLY, Composition, Quantification, …)
  - Sensitive to variable ordering
  - [Bryant, 86 & 92]



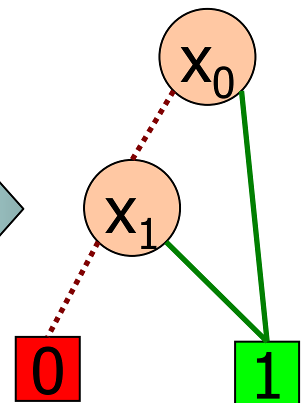


Kripke Structures as BDDs

- Basic idea: use characteristic functions for set representation
  - Each state is uniquely encoded by atomic propositions (state variables)
  - Use BDDs instead of truth table

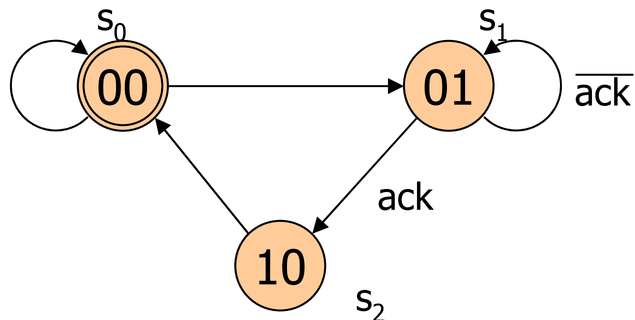
state	x_1	x_0	$\mathcal{S}$
s_0	0	0	1
s_1	0	1	1
s_2	1	0	1
-	1	1	0

$$\mathcal{K}_{\mathcal{S}}(x_0, x_1) = \neg(x_0 \wedge x_1)$$



Kripke Structures as BDDs

- Transitions are encoded by current state, input and next state propositions

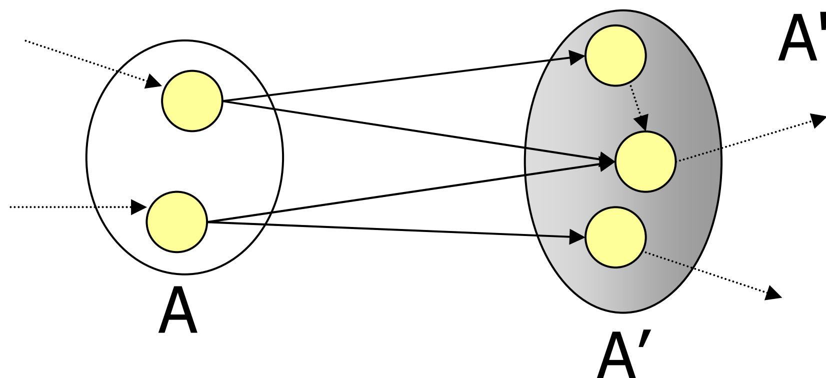


$\mathcal{T}$	x_1	x_0	ack	x_1'	x_0'	
	0	0	-	0	0	1
	0	0	-	0	1	1
	0	1	0	0	1	1
	0	1	1	1	0	1
	0	1	-	0	0	1
....						0

$$\mathcal{K}_{\mathcal{T}}(x_0, x_1, ack, x_0', x_1')$$

Symbolic Model Checking

- Once again
  - Traverse state space
  - Find *good* or *bad* states
- Main operation: Image(A) / PreImage(A)
  - Input: set of states A
  - Output: set of successor (predecessor) states A'



$$A' = \{\, s' \mid \exists s.\exists i.(s, i, s') \in T \wedge s \in A\}$$

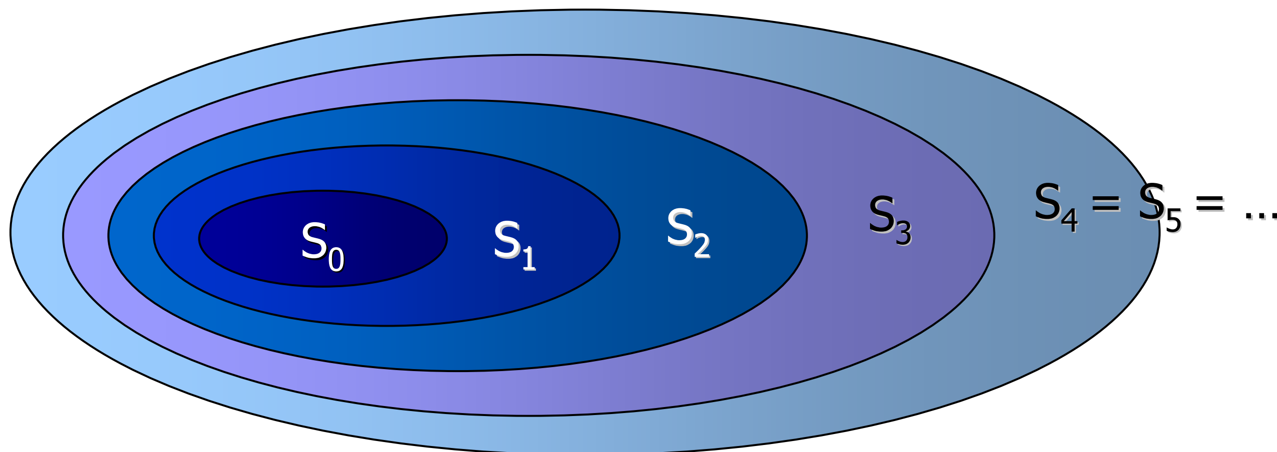
As characteristic function:

$$A' = \exists s.\exists i.(\, \mathcal{K}_{\mathcal{T}}(s, i, s') \wedge \mathcal{K}_{\mathcal{A}}(s)\,)$$

Reachability Analysis

- Fixpoint algorithm

```
Reached = S0; // start with initial state set
do {
        ReachedBefore = Reached;
        Reached = Reached ∪ Image(Reached);
}
while (ReachedBefore != Reached); // fixpoint reached?
```

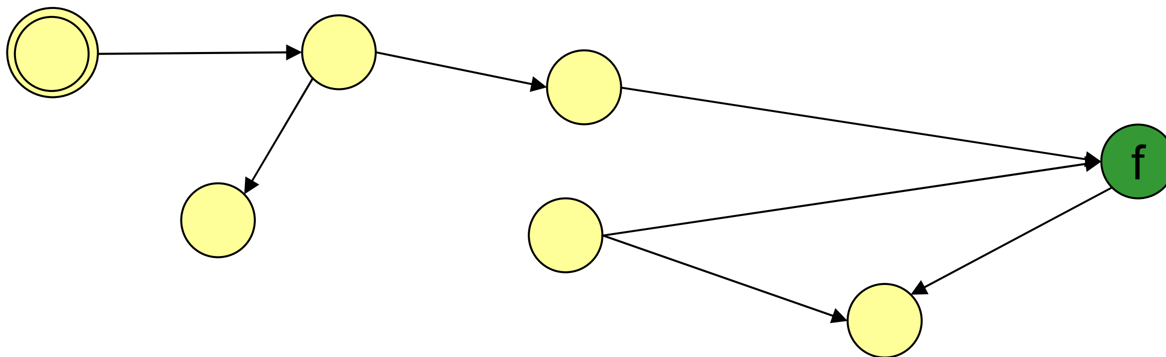


CTL Model Checking

- Formula f can be associated with set of satisfying states → extension set
- Compute extension sets recursively in the syntax graph of the formula
- Check $S_0 \subseteq ext(f)$
- Represent extension sets by BDDs
- Use state space traversal for temporal operators

Example: EF Operator

- All states having a path to f hold **EF** f




```
current = ext(f); old = ∅;
while (old ≠ current) { // fixpoint reached?
  old = current;
  current = old ∪ PreImage(current);
}
```

Model Checking – State of the Art

- Fixpoint iterations for property proving
- NuSMV
  - Reimplementation of classical model checker SMV
  - BDD & SAT engines
- BLAST, Bandera, Java Pathfinder, Murphy, Rulebase, SLAM, SMV, …
- Optimizations (ongoing research)
  - Abstraction
  - Modularization
  - Partial order reduction
  - Early quantification
  - Partition transition relation
  - Alternative representations (MTBDD, BED, …)

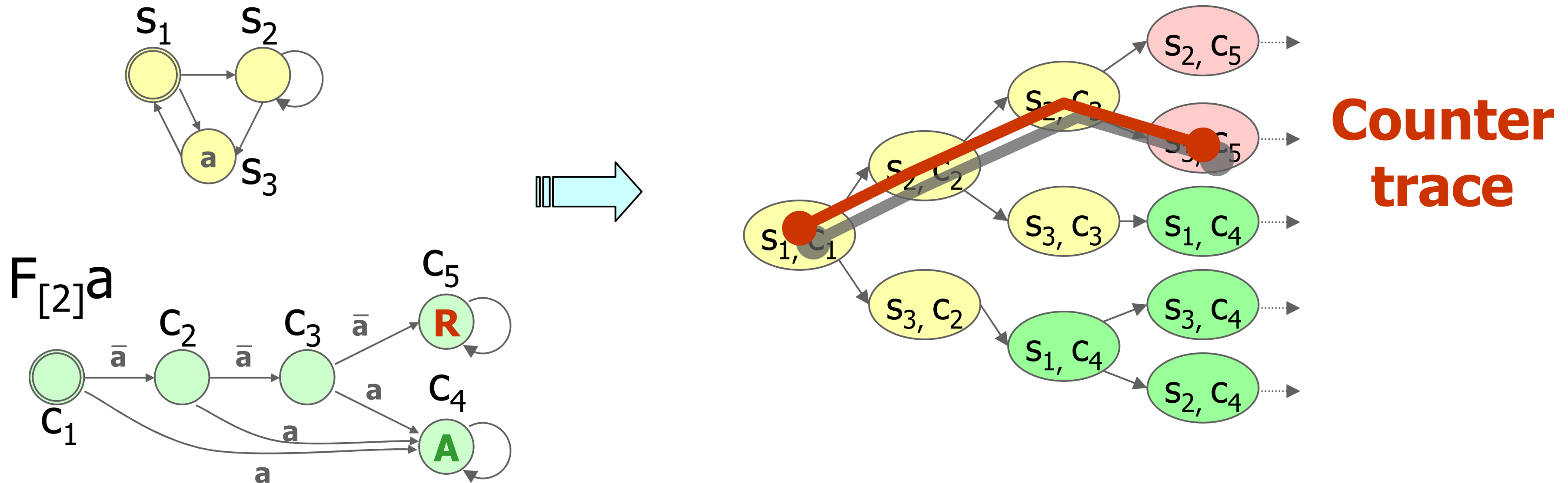
Alternative Approach: SymC

- Bounded model checking (bmc)
  - Check properties for limited number of steps
  - Reduce to SAT instance

- Bounded property checking with BDDs
  - Implemented with SymC tool
  - Keep only current state set (no fixpoint iteration)
  - Reason about traces (universally and existentially)
  - Also based on AR-Automata
  - → Symbolic traversal of system model

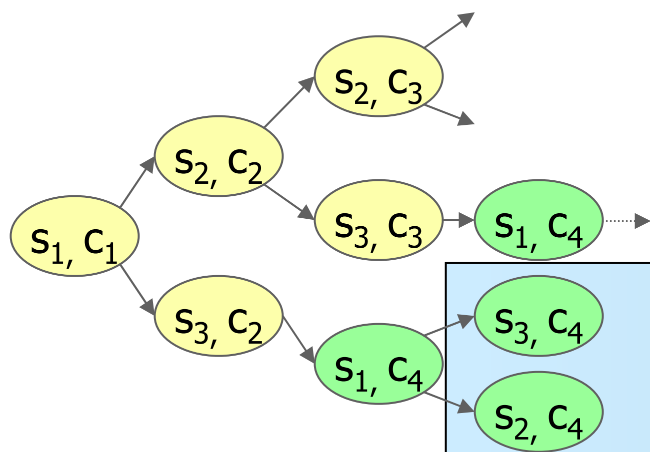
Traversal Strategy in SymC

- Symbolically simulate the product machine
  - Model and AR-automata
  - Finding R-state: formula universally disproved
  - Finding A-state: formula existentially proved



SymC Optimizations

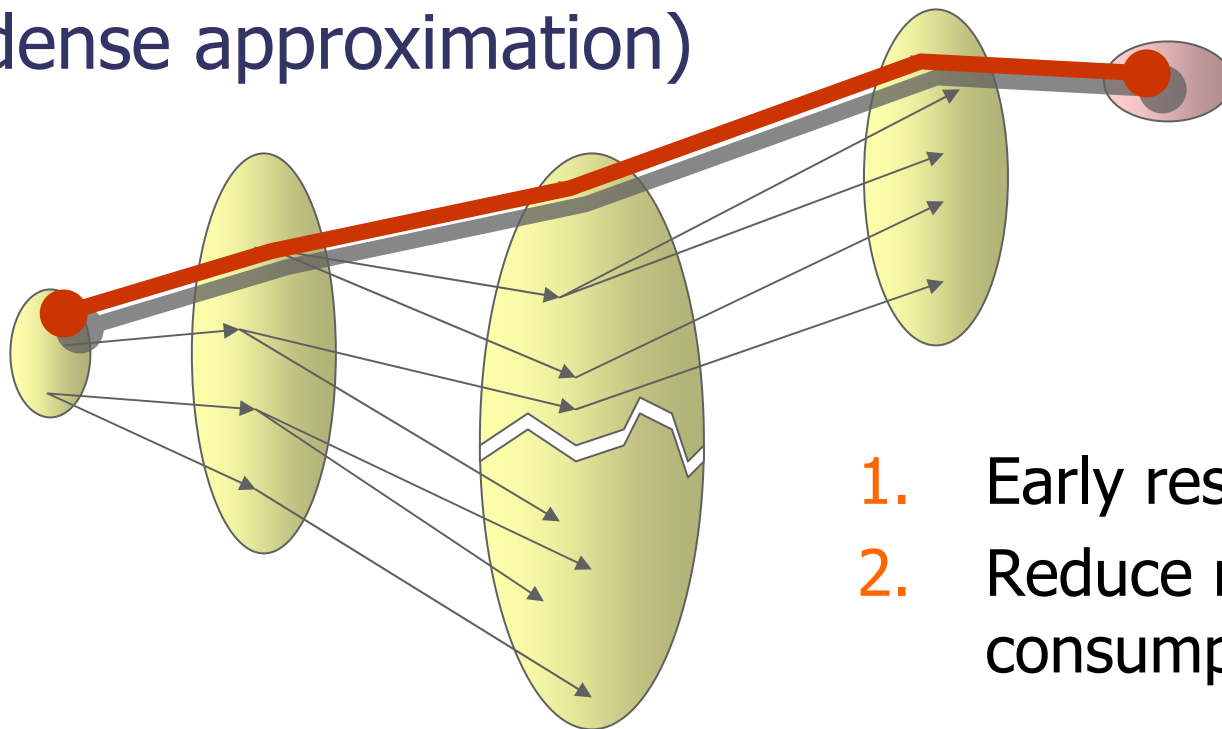
- Standard optimizations: cone of influence, …
- **Pruning**
  - Remove accepted states
  - Pending states are further traversed



Successors of accepted states stay accepted
➔ remove from simulation

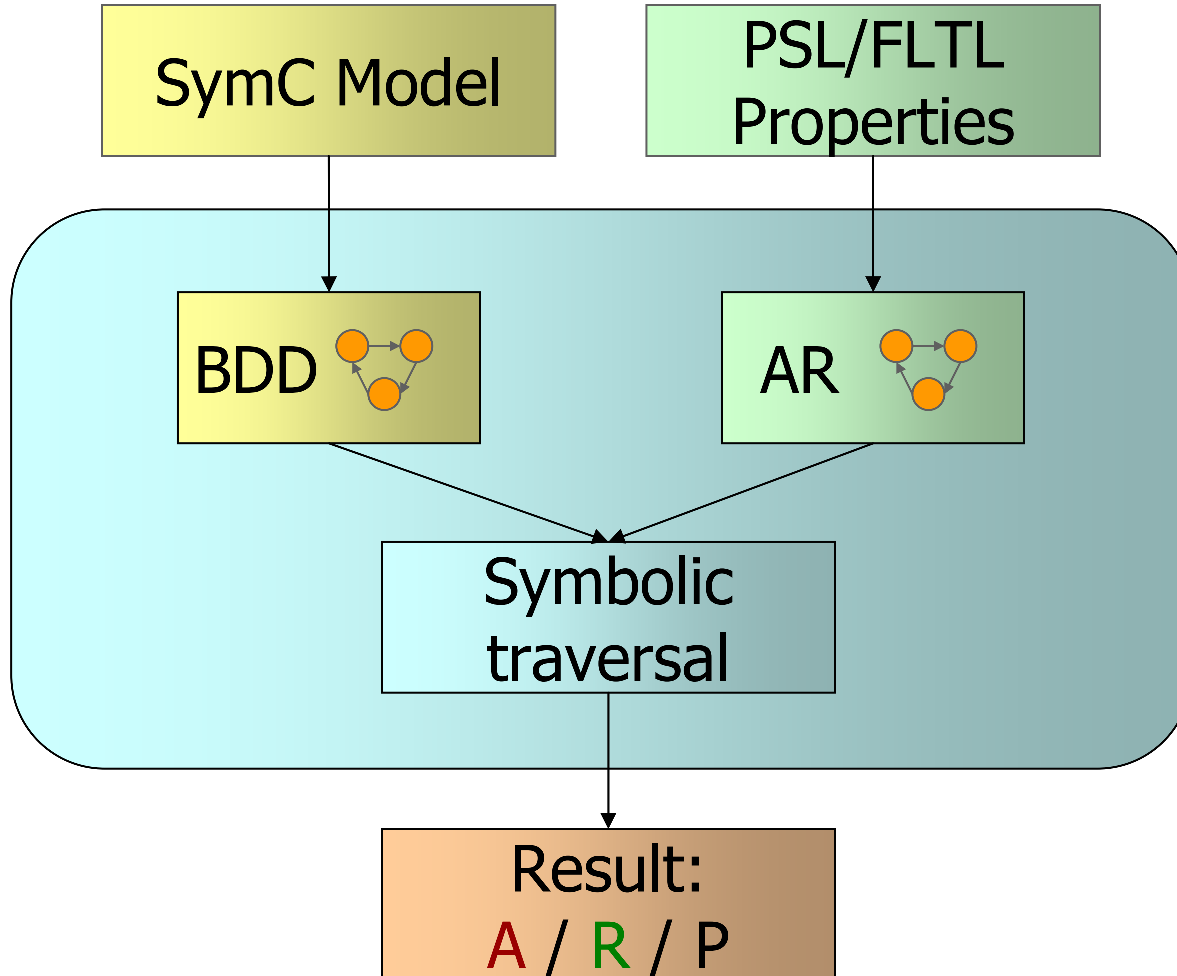
SymC Optimizations

- **Splitting**
  - Split state set and handle partitions separately
  - Split performed with BDD standard algorithm (dense approximation)



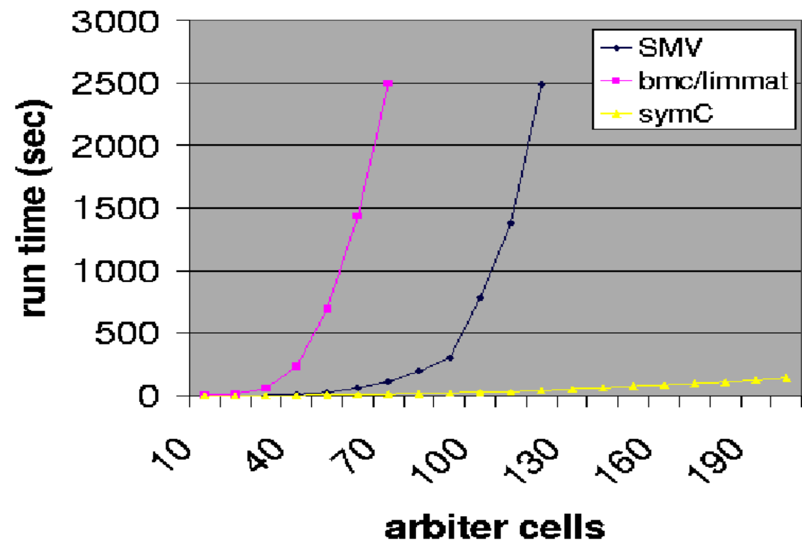
1. Early result detection
2. Reduce memory consumption

SymC Property Checking



SymC - Results

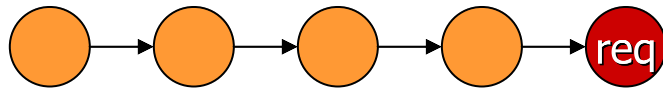
- Well suited for long traces into designs (large property bounds)
- Outperforms SAT tools for certain models/props
- [Ruf et al., FDL'03]

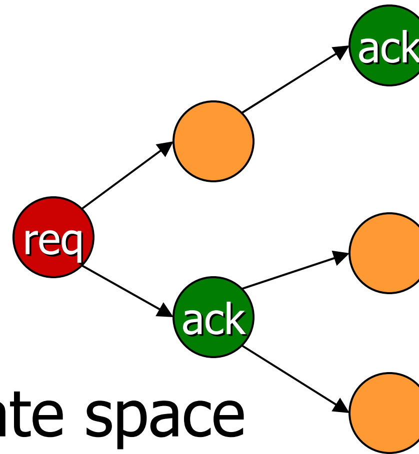


- Future work
  - Parallel version for machine clusters
  - Integration of SAT engine

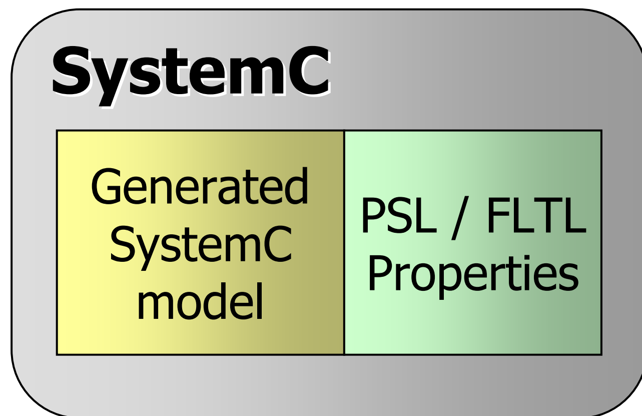
Simulation & Property Checking

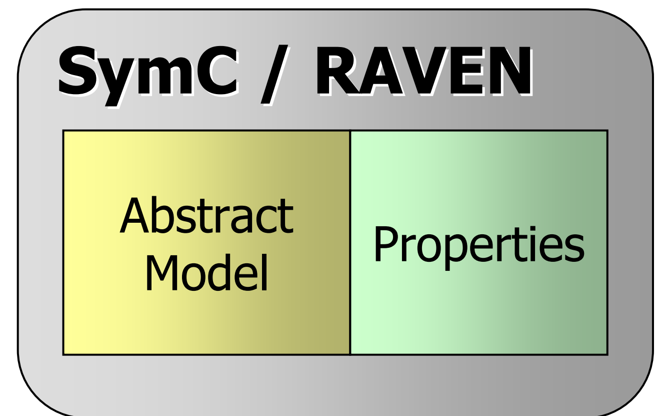
simulation with assertions





local state space traversal





Conclusions

- Formal property specification…
  - Enhances design flow
  - Can be reused in various phases of verification
- Property checking…
  - Enhances system reliability
  - Speeds up development
- Future directions
  - Combine functional and formal approaches to verification
  - Combine related technologies (BDD, SAT)
  - Abstraction (automatic, semiautomatic)
  - …

➔ Verification major research issue

Thank you!

Questions?