The Quest for Correctness: Beyond Verification

University of Athens, October 3 2008

Joseph Sifakis
VERIMAG Laboratory

Correctness by checking vs. Correctness by construction

Building systems which are correct with respect to given requirements is the main challenge for all engineering disciplines

Correctness can be achieved:

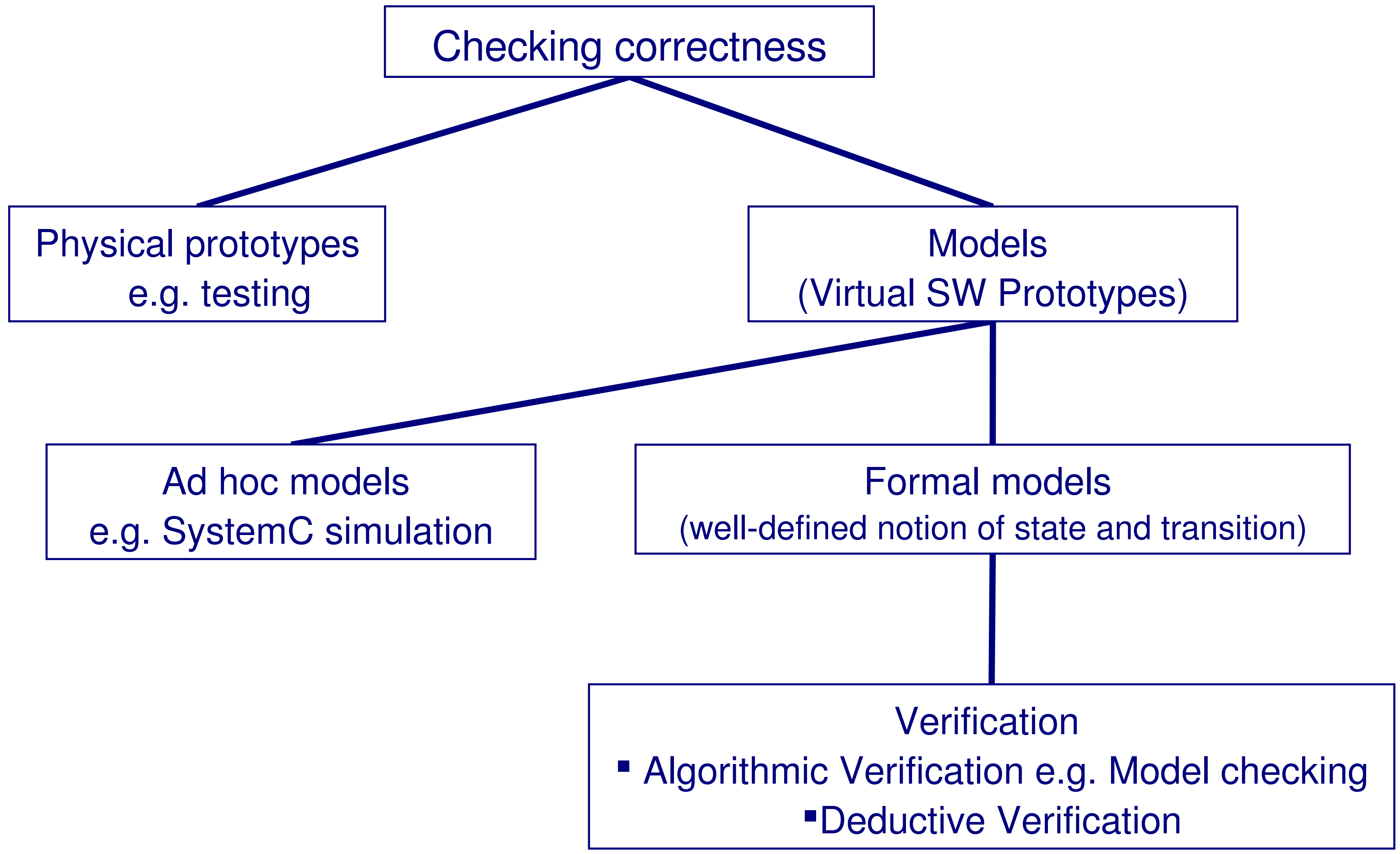
- Either by checking that a system or a model of a system meets given requirements
- Or by construction by using results such as algorithms, protocols, architectures e.g. token ring protocol, time triggered architecture

A big difference between Computing Systems Engineering and disciplines based on Physics is the importance of *a posteriori* verification for achieving correctness

OVERVIEW

- Current status
- Work directions
- Conclusion

Approaches for checking correctness


Checking correctness
Physical prototypes e.g. testing
Models (Virtual SW Prototypes)
Ad hoc models e.g. SystemC simulation
Formal models (well-defined notion of state and transition)
Verification
▪ Algorithmic Verification e.g. Model checking
▪Deductive Verification

Verification: Three essential ingredients

- **Requirements**
  describing the expected behavior, usually as a set of properties

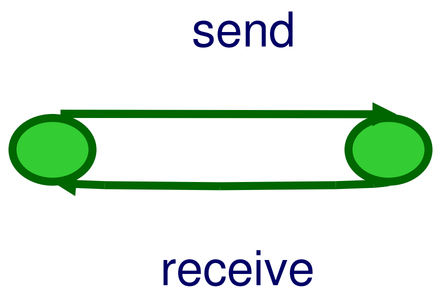
- **Models**
  describing a transition relation on the system states

- **Methods**
  for checking that the models satisfy the requirements

Requirements specification (1/3)

State-based

Using a machine (monitor) to specify observable behavior



Good for characterizing causal dependencies e.g. sequences of actions

Property-based

Using formulas, in particular *temporal logic*, to characterize a set of execution structures e.g. traces, execution trees

always(inev (enable(send)))

always(inev (enable(receive)))

Good for expressing global properties such as mutual exclusion, termination, fairness

Requirements specification (2/3)

About Temporal logic [Pnueli, Lamport, Clarke & Emerson]

This was a breakthrough in understanding and formalizing requirements for concurrent systems. Writing rigorous specifications in temporal logic is not trivial.

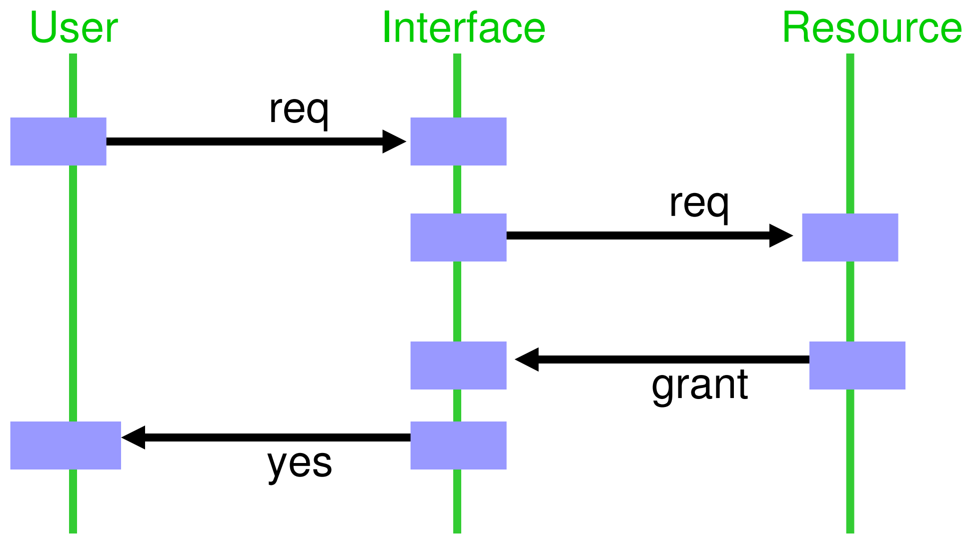
- There exist subtle differences in the formulation of common concepts such as liveness and fairness depending on the underlying time model   e.g. always(inevitable(f))

- The declarative and dense style in the expression of property-based requirements is not always easy to master and understand. Are specifications
    - **Sound**: there exists a model satisfying it
    - **Complete**: tight characterization of system behavior

Pragmatically, we need a combination of both property-based and state-based styles, e.g. PSL

Requirements specification (3/3)

Moving towards a “less declarative” style by using notations such as MSC’s or monitors closer to state-based specifications.



Much to be done for extra-functional requirements characterizing:

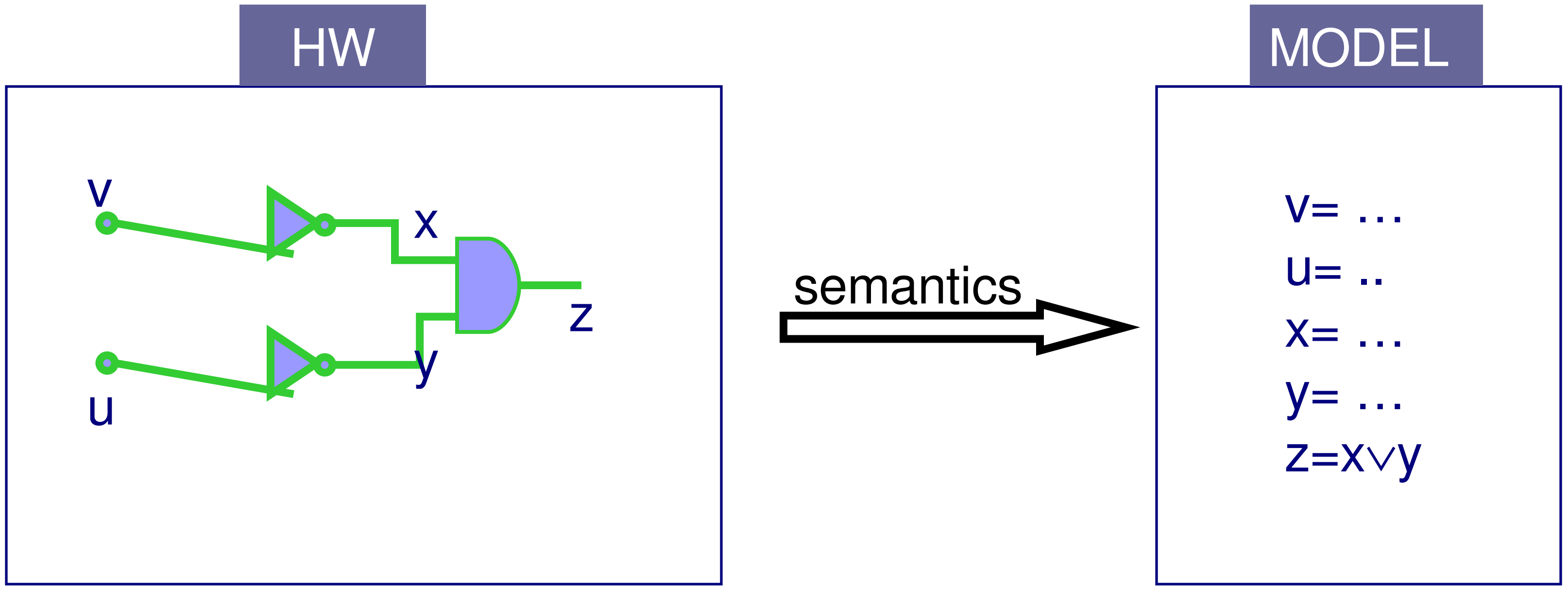
- security (e.g. privacy properties),
- reconfigurability (e.g. non interference of features),
- quality of service (e.g. jitter).

Building models (1/3)

Models should be:

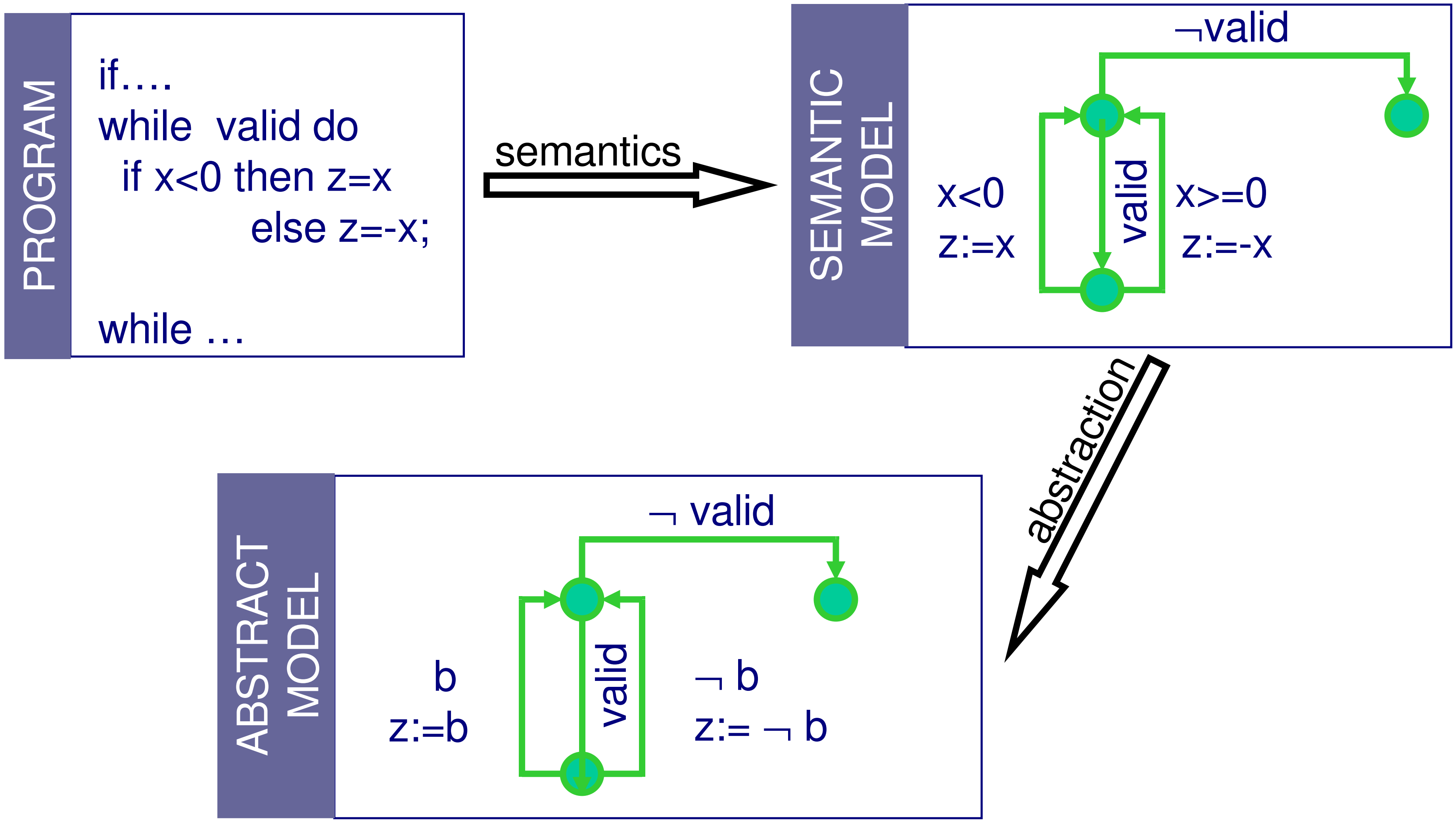
- **faithful** *e.g. whatever property we verify for the model holds for the real system*
- generated **automatically** from system descriptions

For hardware, it is easy to get faithful logical finite state models represented as systems of boolean equations



Building models (2/3)

For software this may be much harder ....



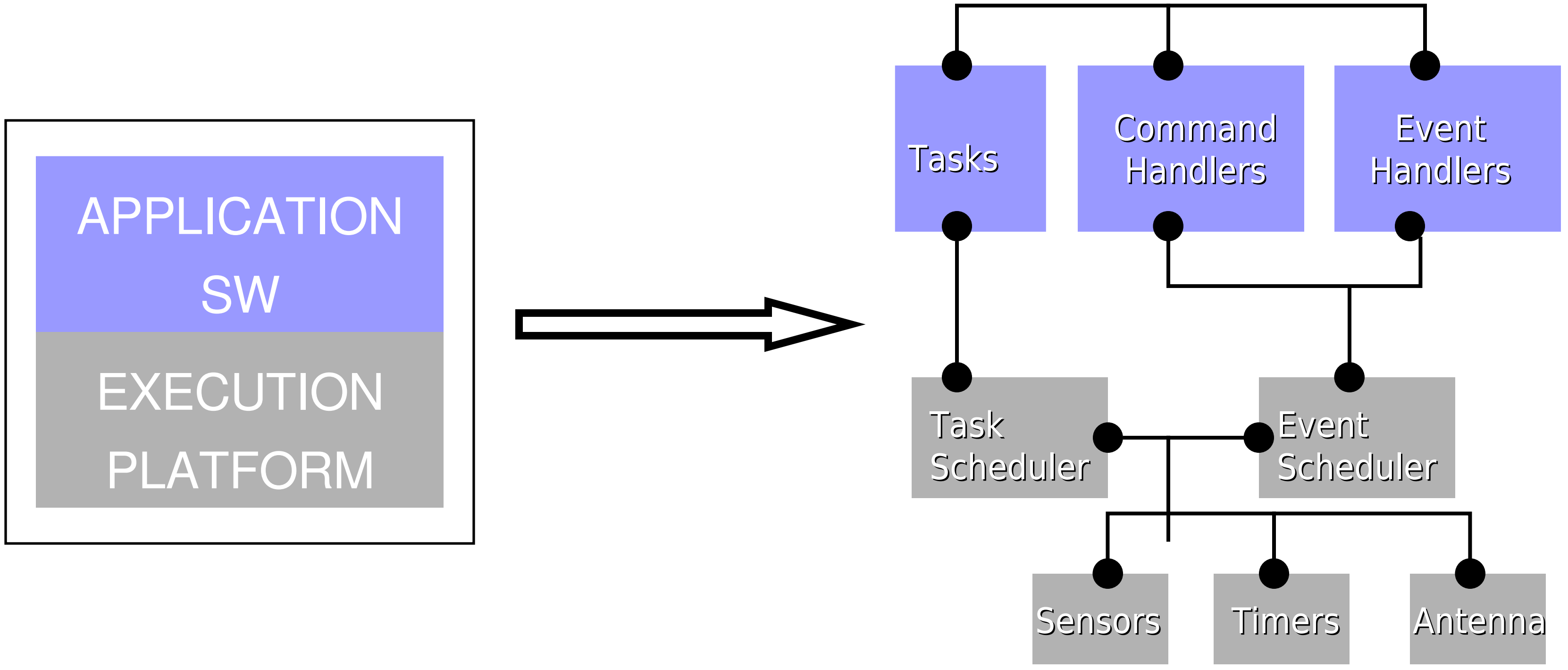
```
if….
while  valid do
  if x<0 then z=x
            else z=-x;

while …
```

Building models (3/3)

For mixed Software / Hardware systems:

- there are no faithful modeling techniques as we have a poor understanding of how **software** and the underlying **platform** interact
- validation by testing physical prototypes or by simulation of ad hoc models



Verification: Deductive vs. Algorithmic (1/2)

Deductive verification

- Based on sets of inference rules for reasoning on the structure of the systems
- General and interactive, targeting verification of infinite state systems, assisted by theorem provers
- Frameworks for the development of deductive proofs: VDM, Z, B

Algorithmic verification

- Based on the analysis of global models obtained by flattening system structure e.g. transition systems
- Emphasis on automation rather than generality
- Main representatives:
  - model checking and
  - abstract interpretation

Verification: Deductive vs. Algorithmic (2/2)

A main idea in the 1980s was to combine **deductive** (human-driven) and **algorithmic** (automated) verification methods e.g.

- model checking to check properties of parts of a complex system,
- deductive techniques to carry out less automatable tasks

The results for such combinations were below expectations.

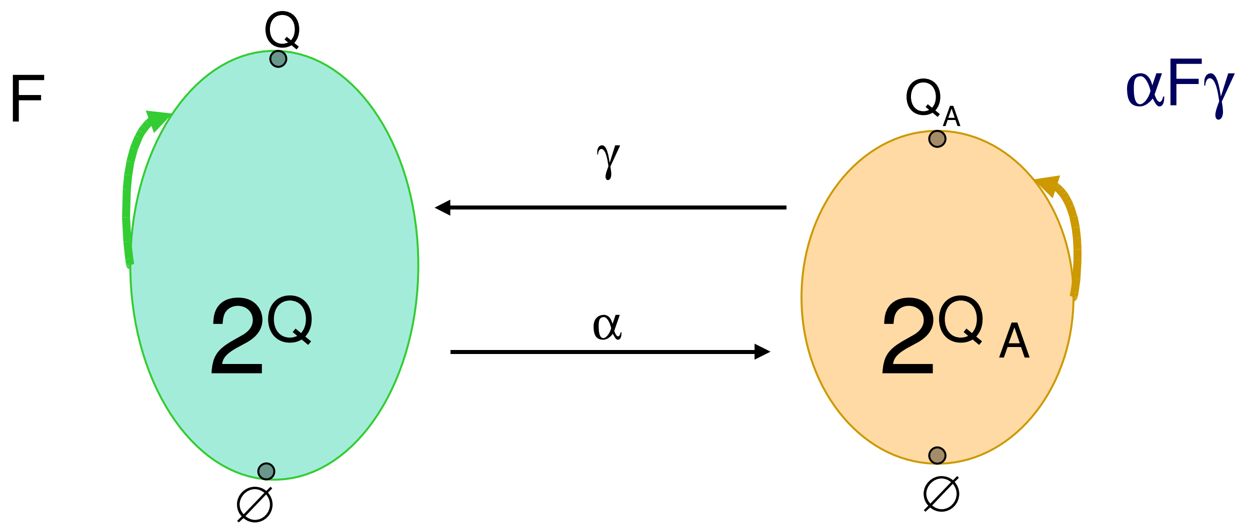
The main obstacle is the lack of effective interaction between automated verification tasks and proofs driven by humans, who:

- can handle only data of limited complexity
- can be inventive only if they have a global understanding of the reasoning process

Algorithmic Verification: Using Abstraction (1/2)

S_A satisfies f_A implies S satisfies f
where $S_A = (Q_A, R_A)$ is an **abstraction** of $S=(Q,R)$
for formulas f involving only universal quantification over execution paths

[Cousot&Cousot 79] **Abstract interpretation,** a general framework for computing abstractions based on the use of Galois connections



- α,γ are monotonic
- $Id \subseteq \gamma\alpha$
- $\alpha\gamma \subseteq Id$

αFγ is the best approximation of F in the abstract state space

Algorithmic Verification: Using Abstraction (2/2)

Model checking

- Initially, focused on finite state systems (hardware, control intensive reactive systems).
  Later, it addressed verification of infinite state systems by using abstractions.

- Used to check general properties specified by temporal logics.

Abstract interpretation

- Driven by the concern for finding adequate abstract domains for efficient verification of program properties, in particular properties related to program execution.

- Focuses on forward or backward reachability analysis for specific abstract domains.

Significant results can still be obtained by combining these two approaches

e.g. by using libraries of abstract domains in model checking algorithms.

- Current status
- Work directions
- Conclusion

O
V
E
R
V
I
E
W

Work directions: Component-based modeling

Develop theory and methods for building faithful models for mixed SW/HW systems as the composition of heterogeneous components

Sources of heterogeneity

- Abstraction levels: hardware, execution platform, application software
- Execution: synchronous and asynchronous components
- Interaction: function call, broadcast, shared memory, message passing etc.

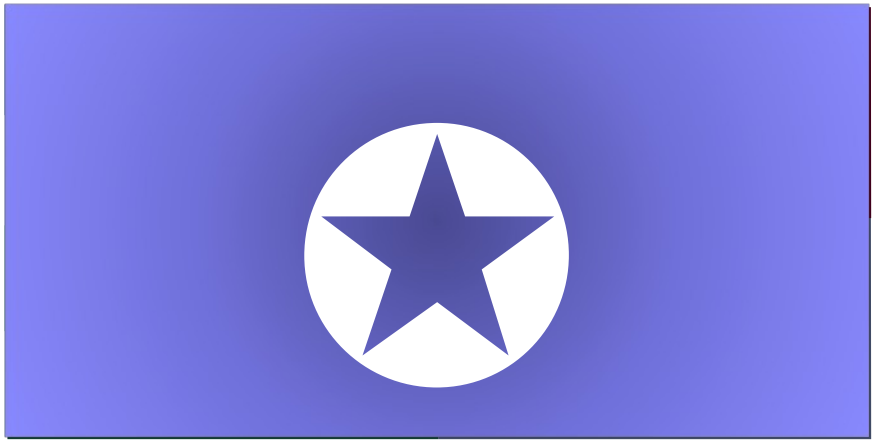
We need to move

from	low level automata-based composition
to	a unified composition paradigm encompassing architecture constraints such as protocols, schedulers, buses.

Work directions: Compositional verification

roving properties of a composite component from properties of

- individual components
- its architecture



We need to move

	Composition operation	Properties
from	Automata-based	Safety, liveness
to	Component-based	Specific properties e.g. Deadlock-freedom, mutex

Work directions: Compositional verification

Develop compositionality results

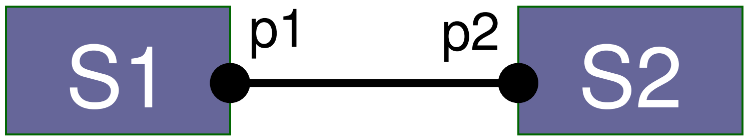
- For particular
  - architectures (e.g. client-server, star-like, time triggered)
  - programming models (e.g. synchronous, data-flow)
  - execution models (e.g. event triggered preempable tasks)
- For specific classes of properties such as deadlock-freedom, mutual exclusion, timeliness

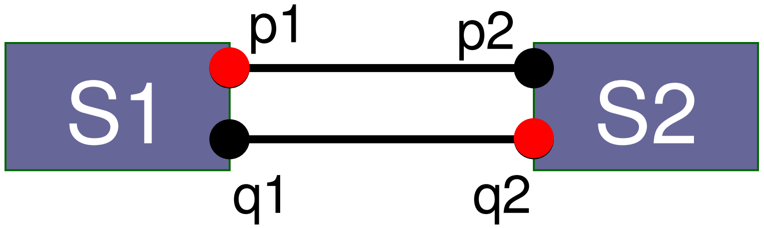
Compositionality rules and combinations of them lead

- to “verifiability” conditions, that is conditions under which verification of a particular property becomes much easier.
- to correct-by-construction results

Work directions: Compositionality - example

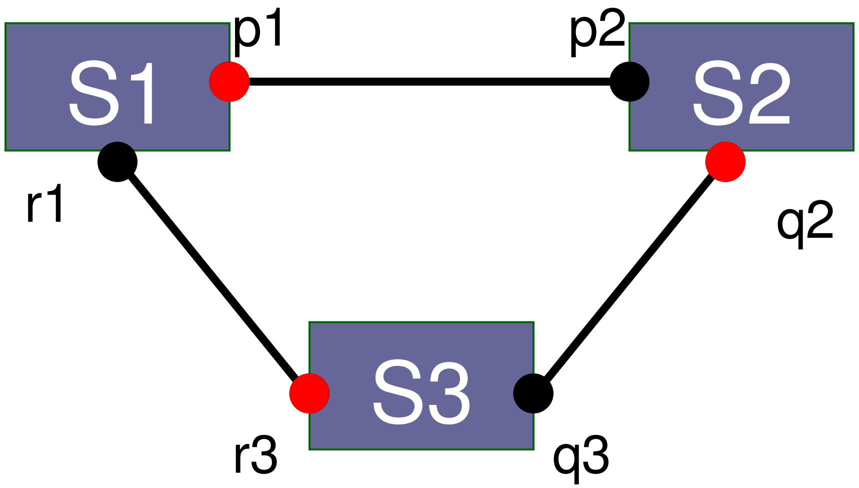
Checking global deadlock-freedom of a system
built from deadlock-free components,
by separately analyzing the components and the architecture.





Potential deadlock

$D = en(p1) \wedge \neg en(p2) \wedge en(q2) \wedge \neg en(q1)$



Potential deadlock

$D = en(p1) \wedge \neg en(p2) \wedge en(q2) \wedge \neg en(q3) \wedge en(r3) \wedge \neg en(r1)$

Work directions: Compositionality - example

Eliminate potential deadlocks D by checking that

$\mathbf{I \wedge D = false}$

where I is a global invariant computed compositionally

Example	Nb Comp	Nb Ctrl St	Nb Bool Var	Nb Int Var	Nb Pot Deadl	Nb Rem Deadl	time
Temperature Control (2 rods)	3	6	0	3	8	8	3s
Temperature Control (4 rods)	5	10	0	5	32	15	1m05s
UTOPAR (4 cars,9 CU)	14	45	4	26	??	0	1m42s
UTOPAR (8 cars,16 CU)	25	91	8	50	??	0	22m02s
R/W (50 readers)	52	106	0	1	~10^15	0	1m15s
R/W (100 readers)	102	206	0	1	~10^30	0	15m28s
R/W (130 readers)	152	266	0	1	~10^39	0	29m13s

Results obtained by using the D-Finder tool: http://www-verimag.imag.fr/~thnguyen/tool/

- Current status
- Work directions
- Conclusion

O
V
E
R
V
I
E
W

From a posteriori verification to constructivity at design time

Verification is not the only way for guaranteeing correctness.

- In contrast to Physics, Computer Science deals with an infinite number of possibly created universes
- Limiting the focus on particular tractable universes of systems can help overcome current limitations

We should concentrate on compositional modeling and verification for sub-classes of systems and properties which are operationally relevant and technically successful

This vision can contribute to the unification of the discipline, by bridging the gap between Formal Methods and Verification, and Algorithms and Complexity.

EUCARISTW