MCI **M**ads **C**lausen **I**nstitute
www.mci.sdu.dk

UNIVERSITY OF SOUTHERN DENMARK

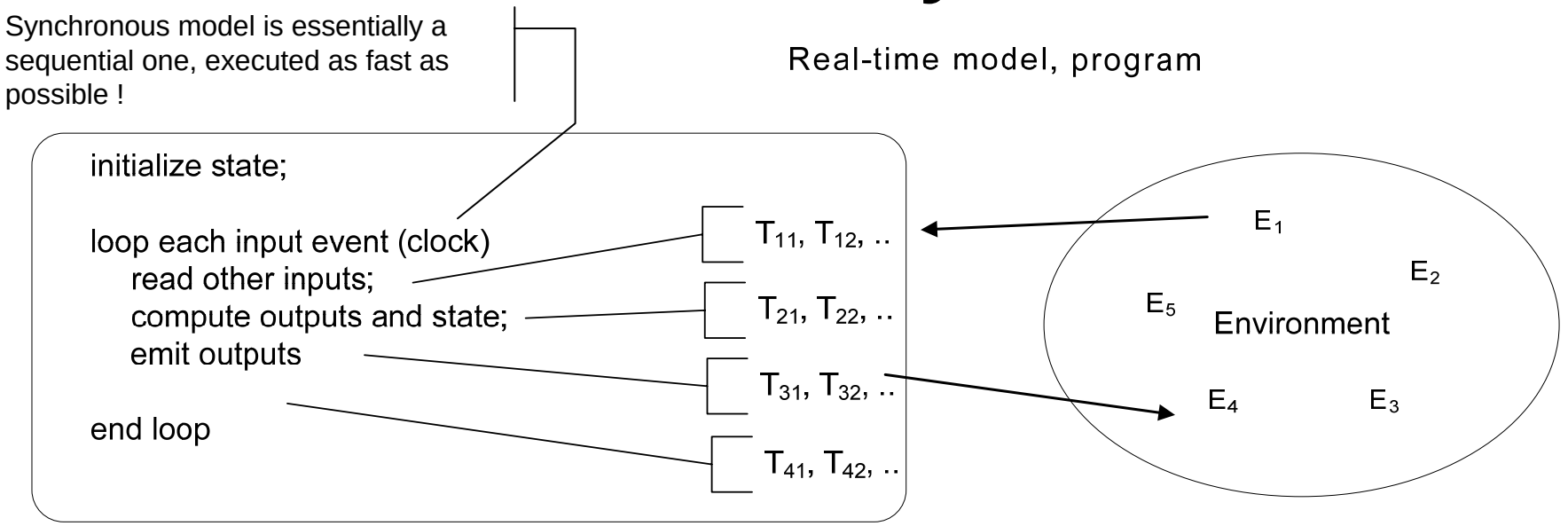
Minimization of the verification effort

Nicolae Marian, MCI
nicolae@mci.sdu.dk

Outline

- Verification: proof of correctness
- Fighting state space explosion:
  abstraction + composition

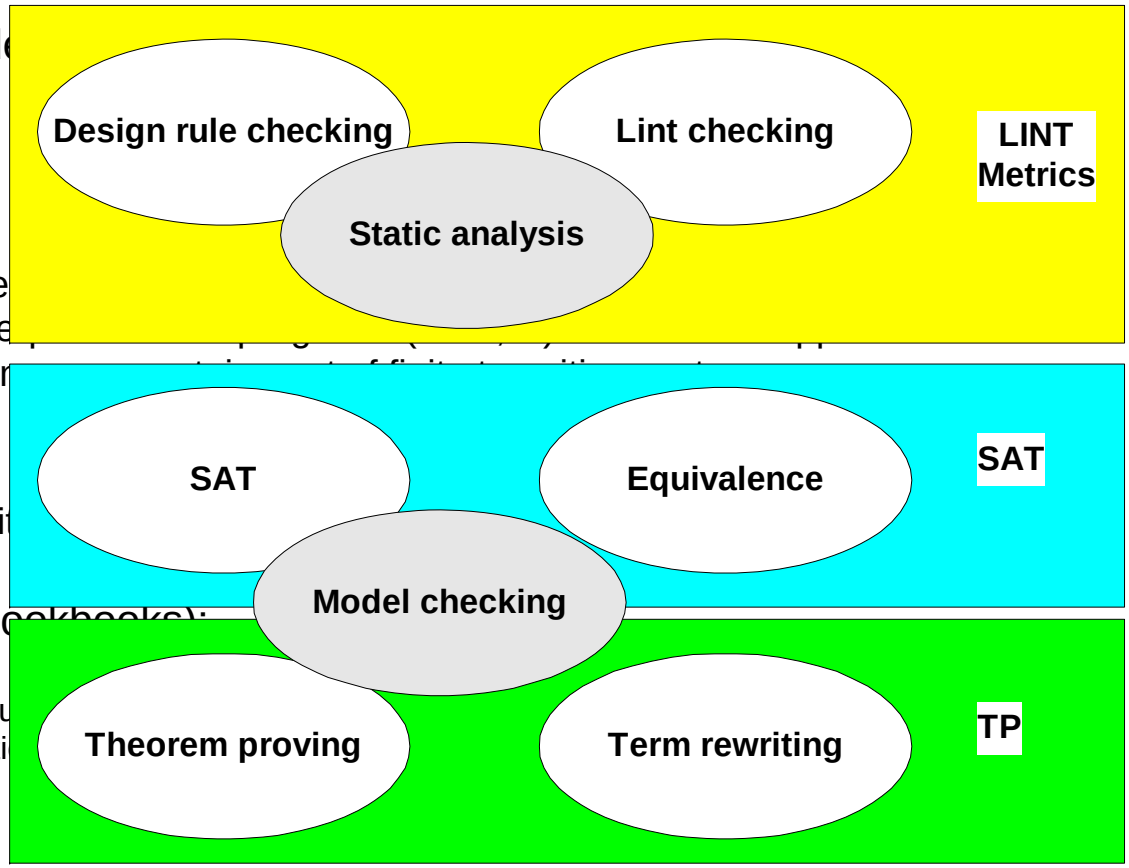
Real-time system



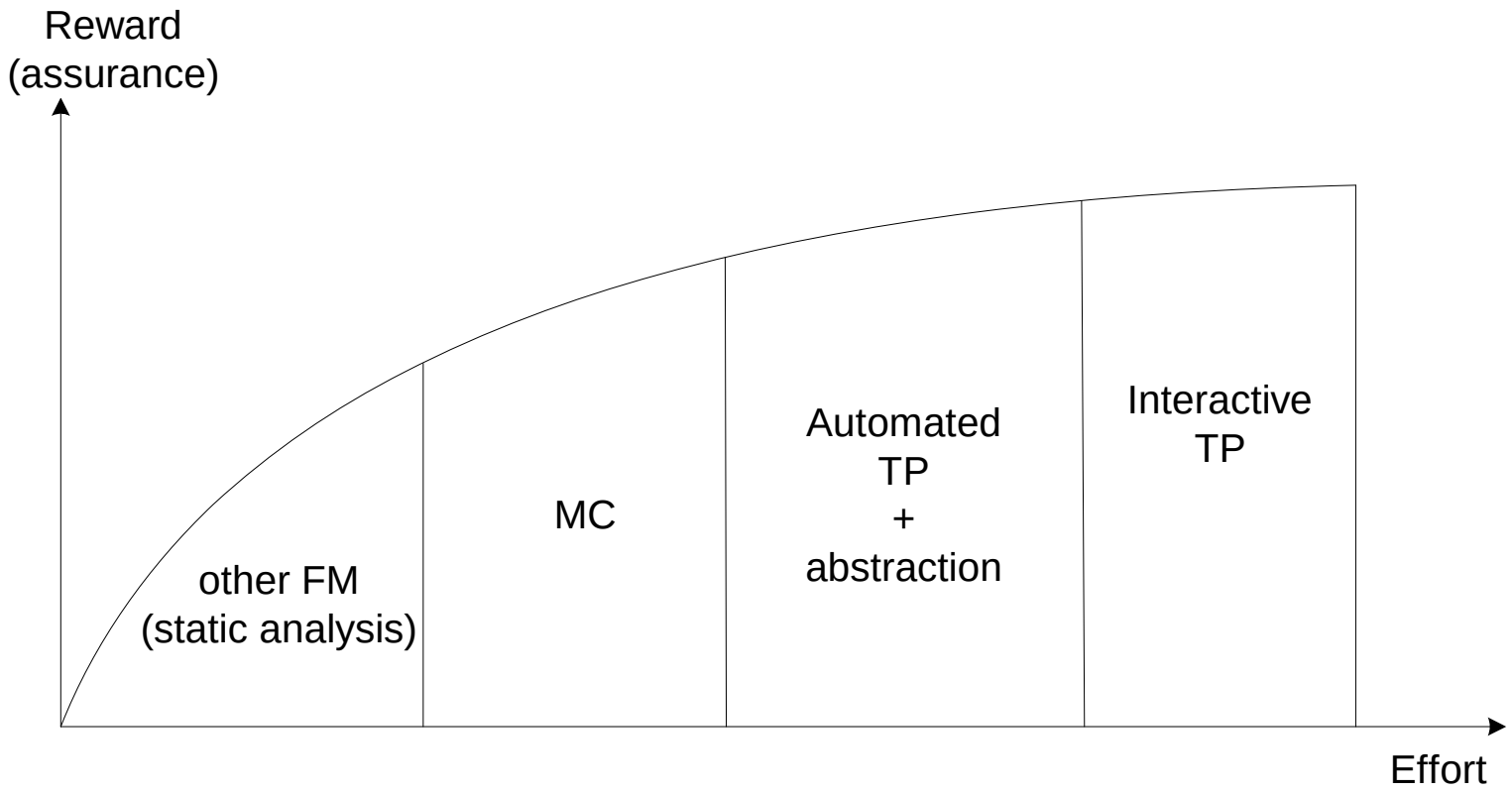
- Questions:
  - How can (sound) abstractions be derived automatically?
  - How can we minimize the verification effort by exploiting abstraction and compositionality (pair/group-wise controller – environment processes)?
  - How can we apply guided (local property, pair/group-wise subsystems) model checking according to start-end chain reactions?

Methods of verification

- Verification by hand - test & d
  - White box - Source code
  - Black box - Executable code
- Algoritmic techniques:
  - Model checking of finite state
  - Static analysis & abstract inte
  - Equivalence, bisimulation, lar
- Deductive techniques:
  - HOL provers for finite & infini
- Combinations of the above (co
  - MC +
    - Symmetry, partial order, bou
    - Abstract interpretation, Stati
    - …
  - SA + TP
  - Verification + testing
  - …

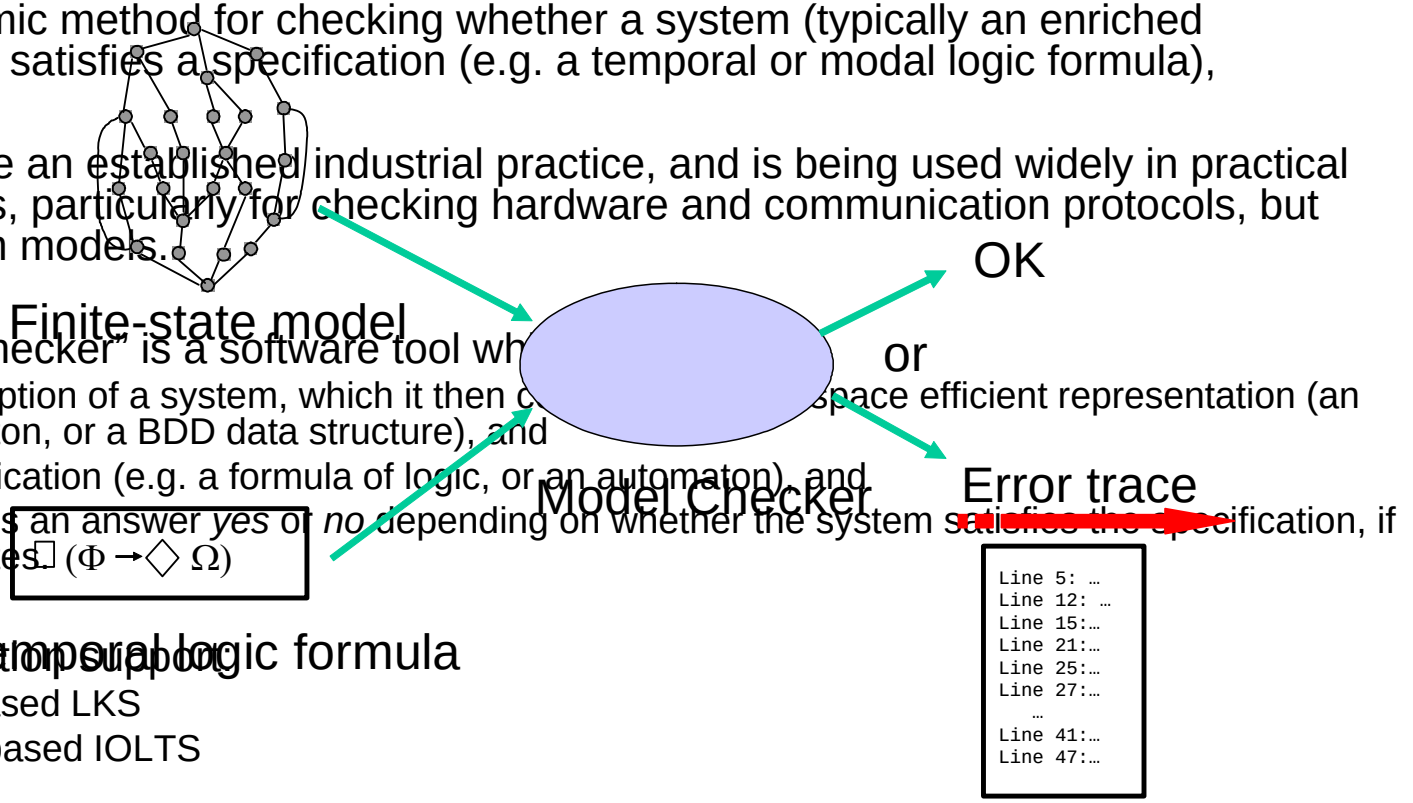


Methods of verification cont'd

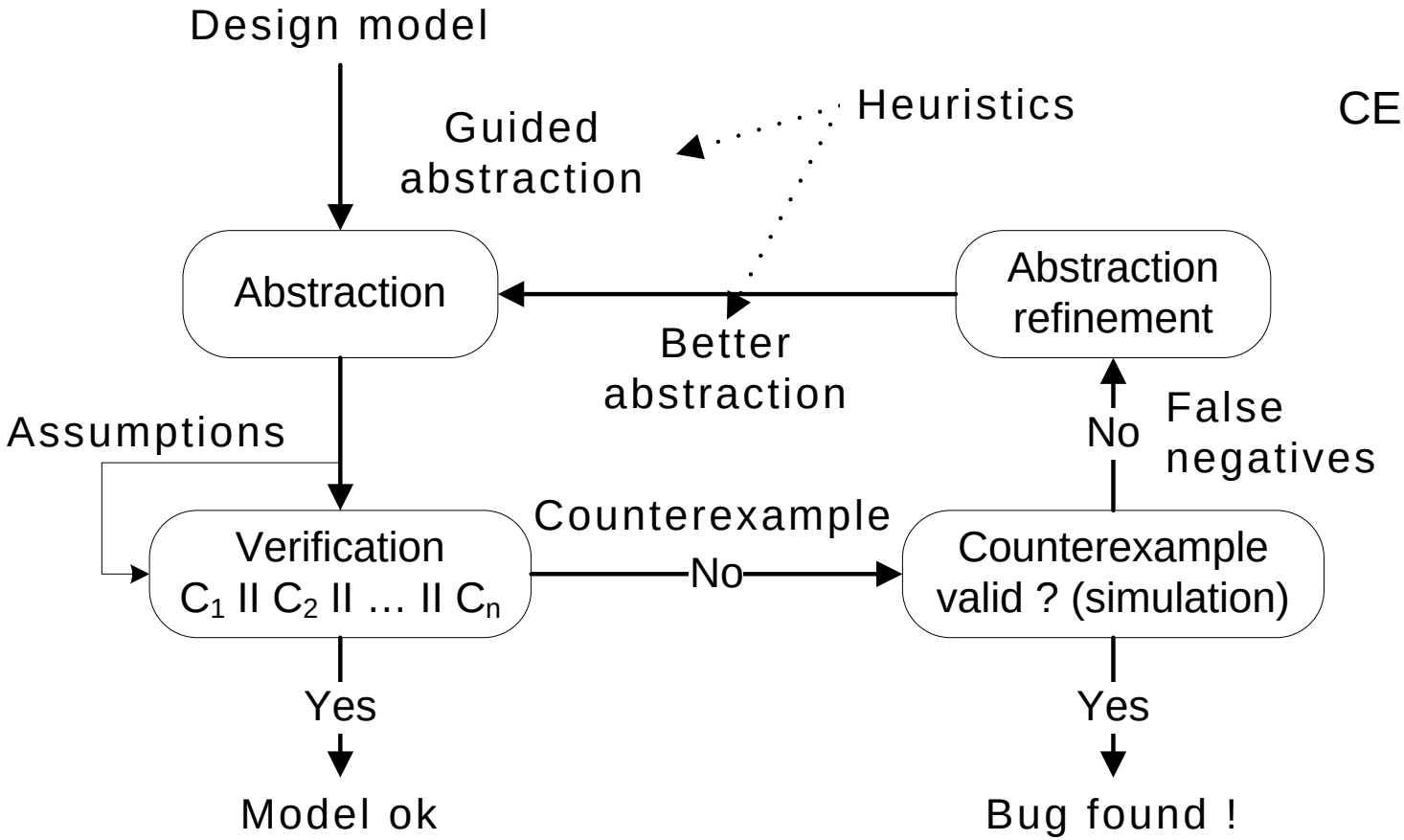


Model checking

- an algorithmic method for checking whether a system (typically an enriched automaton) satisfies a specification (e.g. a temporal or modal logic formula),
- has become an established industrial practice, and is being used widely in practical applications, particularly for checking hardware and communication protocols, but also system models.
- A "model checker" is a software tool wh
  - a description of a system, which it then c space efficient representation (an automaton, or a BDD data structure), and
  - a specification (e.g. a formula of logic, or an automaton), and
  - produces an answer *yes* or *no* depending on whether the system satisfies the specification, if terminates.
- State transition support
  - state-based LKS
  - Action-based IOLTS
- Some problems:
  - counterexamples are not always valid: false negative, spurious counterexample



Optimize MC effort:
Find good abstractions + exploit composition



Theory - practice

- Theory:
  - cut out a sub-component
  - abstract the environment, i.e.,
    - add an environment process which closes the sub-component
      - shows more behavior than the real environment
      - in extremis: add chaos-proces

- Practice (when building the verification model):
  - We have to address interaction between loosely and strongly synchronized components, translation of semantics, e.g. comm, execution, …
  - Types, global (or not) variables
  - Properties are not always decomposable

    $EF(p \sim q) = EFp \sim EFp, \quad \sim \in \{\wedge, \vee\}$ But $EG(p \sim q) \neq EGp \sim EGp$

    $AG(p \sim q) = AGp \sim AGq$ $AF(p \sim q) \neq AFp \sim AFq$

- Deadlock is inherently non-compositional and its absence/presence is not preserved by standard abstractions (when implicit ..)

Complexity

- The cost of algorithms deciding whether S I= φ for some LKS S and temporal formula φ is given in terms of the sizes |S| and |φ| of the inputs.
- Uppaal is optimised for reachability properties

	LTL	CTL
MC	$O(2^f (V+E))$ EXP (formula size) PSPACE COMPLETE	$O(f (V+E))$ LINIAR (formula and model size)
Composition	PSPACE (formula size)	EXP (formula size)

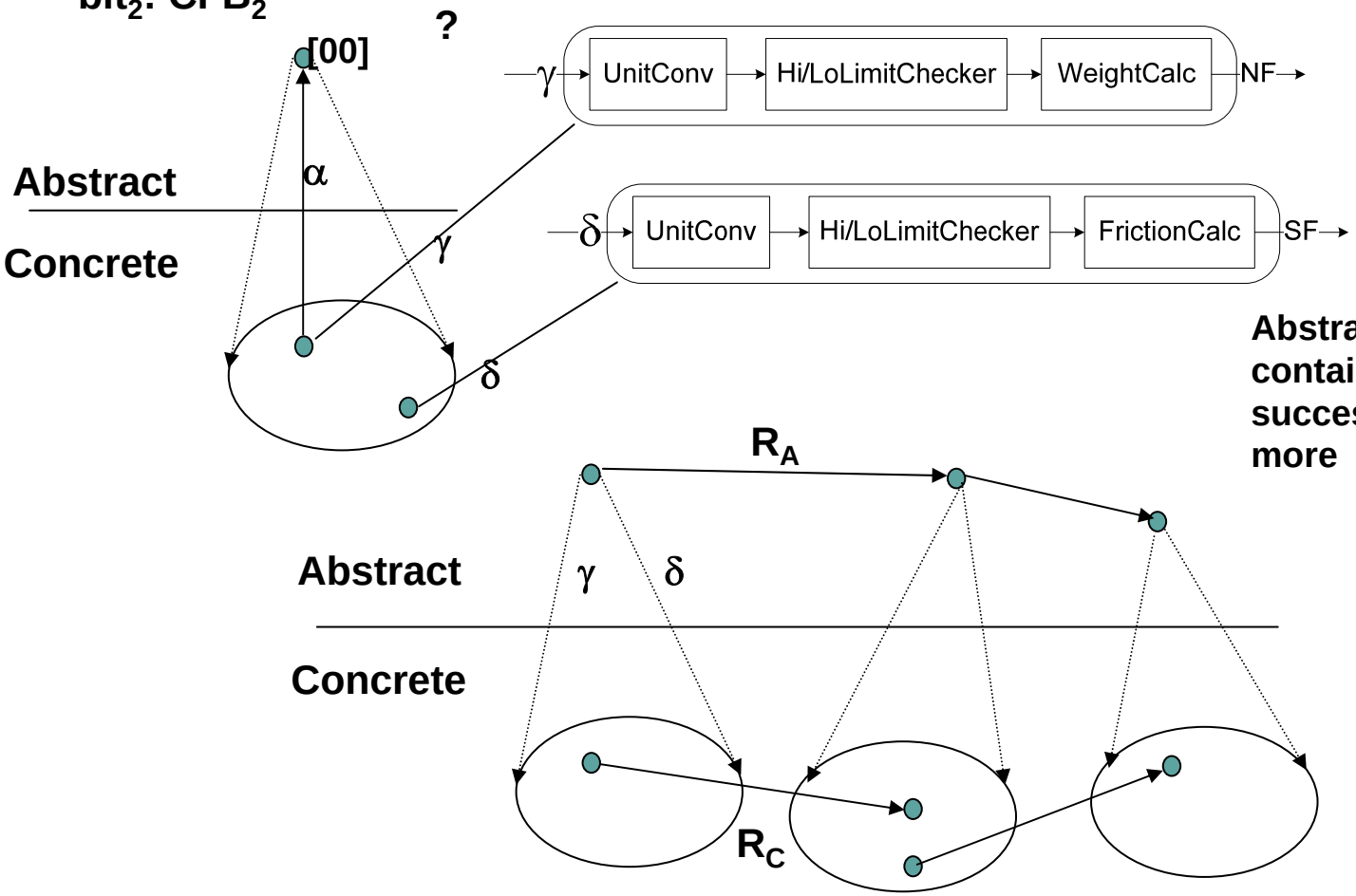
Abstraction

- over-approximations: can get a “false error”, but never a false answer. The abstraction must be safe, i.e. it must contain all possible behaviours of the concrete system.
- data abstraction: replace concrete domains by finite, abstract ones, linear arithmetic dataflow by predicates
- control abstraction, i.e., add non-determinism
  - preserves the relevant behaviors of the system
  - abstractions are most often performed in an informal, manual manner, and require considerable expertise.
- $\Rightarrow$ predicate abstraction
- ingredients: lattice of abstractions, inclusion of behaviours as trace containment or simulation, Tarski’s fixpoint theorem, games in logic and concurrency
  - surjective function h : Concrete $\rightarrow$ Abstract
  - find minimal h (?)
- if abstract model satisfies safety/liveness property $\Rightarrow$ the concrete model does also.
  - Abstraction(Model) I= *Abstraction*(Prop) $\Rightarrow$ Model |= Prop

Abstraction cont'd

Das & Dill

Predicates:
bit_1: CFB_1
bit_2: CFB_2



Abstract next states *must* contain all concrete successors, and possibly more

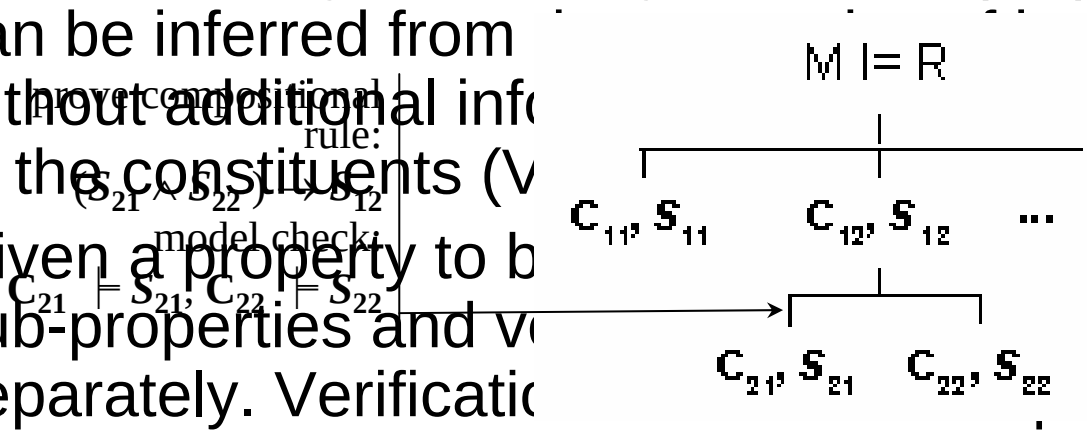
Automatically abstraction generation

- Abtraction techniques [Cousot and Cousot ‘77] [Clarke et al ‘94]
- Localization reduction [Kurshan’95]
- Counterexample Guided Abstraction-Refinement CEGAR

  [Barner et al ‘02] [Chauhan et al ‘02] [Clarke et al ’00 and ‘03] [Das and Dill ‘02] [Glusman et al ‘03] [Gupta et al ‘04] [Jain et al ‘05] [Mang and Ho ‘04] [Wang et al ’01, ’03, ‘04] …

Composition (modularity)

- Definition: Any method by which the property of a system can be inferred from [illegible] constituents, without additional inf[illegible] nternal structure of the constituents (V[illegible]
- Given a property to b[illegible] e property into sub-properties and v[illegible] roperties separately. Verificatio[illegible] ite likely to be restricted to relatively few components and hence would be less expensive than verifying the whole property on the whole system.



Composition cont'd

- Compositional verification of the designs is not sufficiently considered in current tools and might offer better ways to push the limits of verification than addressing the state explosion problem alone.

Compositionality

- Main goal: how to check a system composed of two components, C1 and C2, weather it satisfies a property p, without composing C1 with C2, i.e. C1 II C2 I= p ?

- Solution: non-circular AG

  C1 || A I= P
  C2 I= A

  C1 || C2 I= P

  - Use learning algorithm for regular languages, L*, to automatically generate assumption A
    - How can they be found ? [Namjoshi & Trefler '00]
    - L* learning approach [Giannakopoulou, Pasareanu et al.]

- More general

  C1 I= p1 /\ C2 I= p2 $\Rightarrow$ *CompositionOp*(C1, C2) I= *GlobalPropOp*(p1, p2)
  e.g. *CompositionOp* = II, and *GlobalPropOp* = /\
  C1 I= p1 /\ C2 I= p2 $\Rightarrow$ C1 II C2 I= p1 /\ p2
  C1 $\prec$ C2 $\Rightarrow$ *CompositionOp*(C1, C) $\prec$ *CompositionOp*(C2, C)
  ex. $\prec$ = implementation, refinement;
  C1 $\prec$ C2 $\Rightarrow$ C1 II C $\prec$ C2 II C
  C1 $\prec$ S1 /\ C2 $\prec$ S2 $\Rightarrow$ *CompositionOp*(C1, C2) $\prec$ *CompositionOp*(S1, S2)

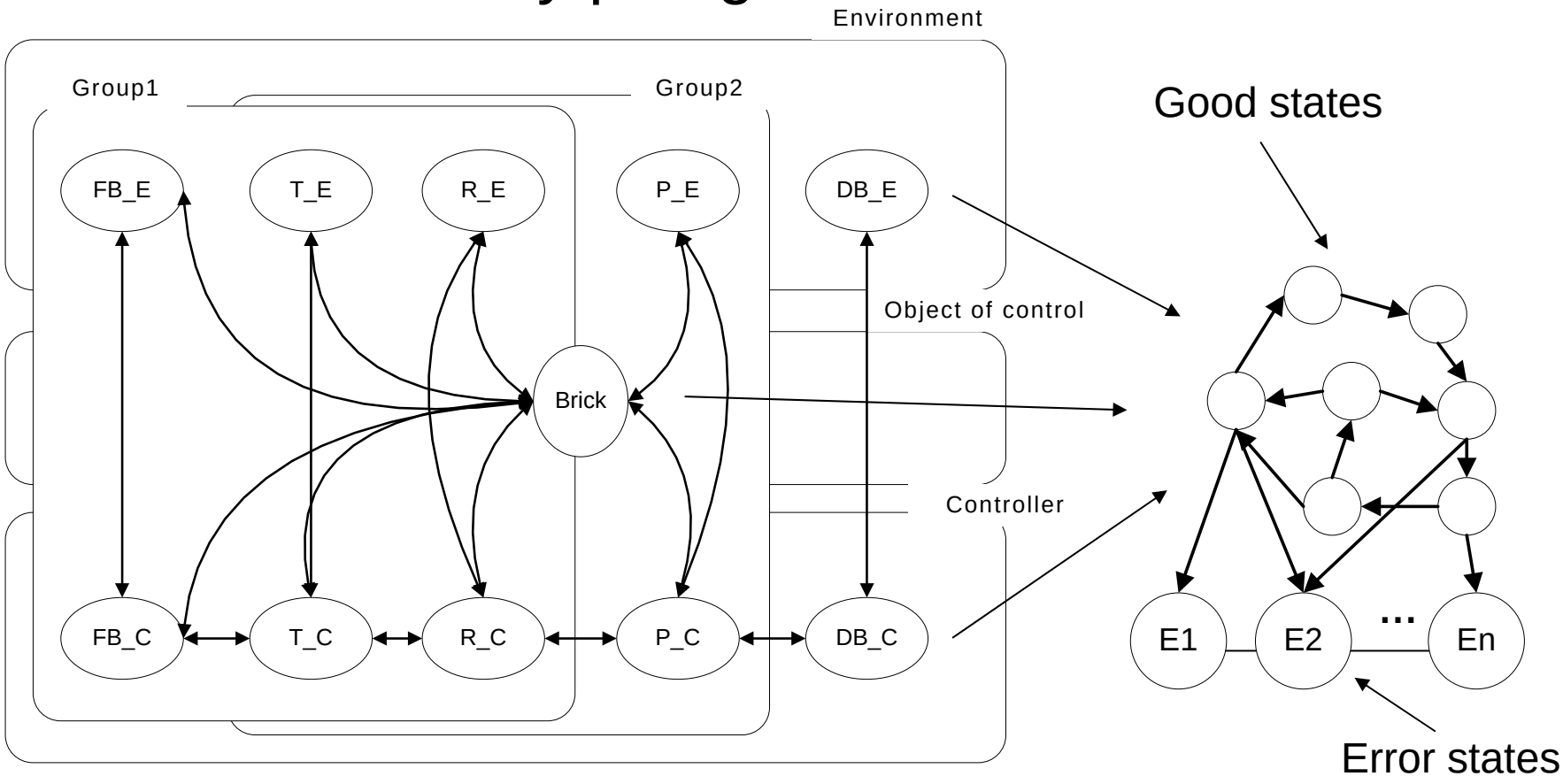
Pros & cons about compositionality

- Composition via conjunction provides a simple but powerful model for reasoning about large systems
- Often, compositional rules are not strong enough.
  - Consider implementations Ii and specifications Si, i = 1, 2. To prove I1 II I2 $\prec$ S1II S2 it would suffice if I1 $\prec$ S1 and I2 $\prec$ S2, but frequently, these individual relations are either satisfied or difficult to prove.
- Components C1 and C2 are not independently designed
  - each relies on functioning in an environment provided by the other

Compositionality cont'd

- Hoare's rule for sequential programs
  - Ideally, we'd like a rule of the form:
  - {P2} C1 {P1}
  - {P1} C2 {P2}
  - {true} C1 || C2 {P1 /\ P2}
  - (C1 guarantees P1 provided that C2 guarantees P2 and vice versa) - is NOT generally sound !
- Circular AG originates with [Chandi & Misra'81, Jones '83] [Pnueli 85] [Clark & Long & McMillan '89] [Abadi & Lamport '91, '95] [Grumberg & Long '94] [de Roever 98] [Barringer 03] (Composing/Conjoining Specifications, Modular Verification)
- Reactive Modules [Alur & Henzinger '95, '99]
- Rule with temporal induction [McMillan'99]
- [Henzinger'01] study of the theory of interfaces
  - formalism focused on components
  - formalism focused on interfaces
- Modularity in Timed and Hybrid Systems [Alur, Henzinger 1997]
  - receptive modules
- Timed Interfaces [de Alfaro, Henzinger, Stoelinga 2002]
- Timed I/O Automata [Kaynar & Lynch, 2003/2004]
- ….

Minimization by pair/group-wise abstractions



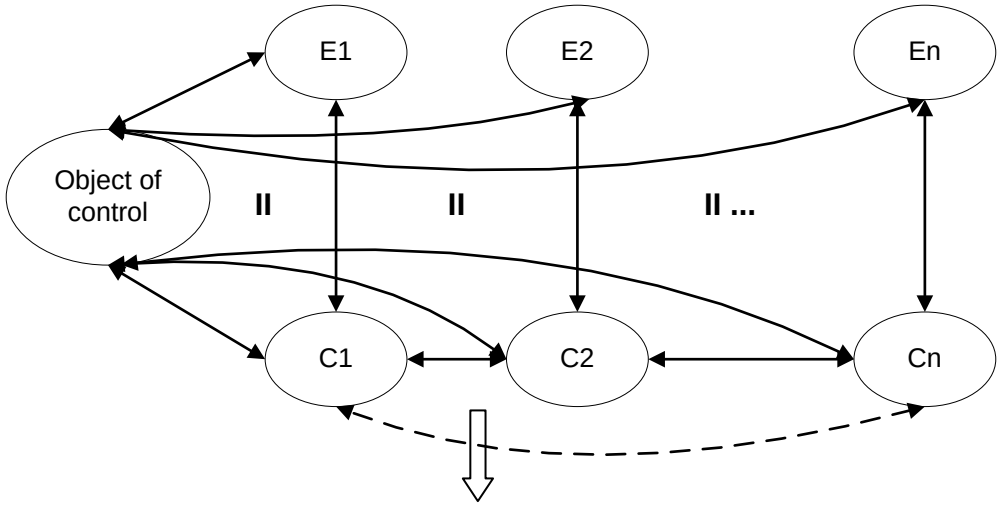
- Augment the system with a π set of explicit “error” states
- Claim: invariants can be expressed in terms of not reaching the π set.

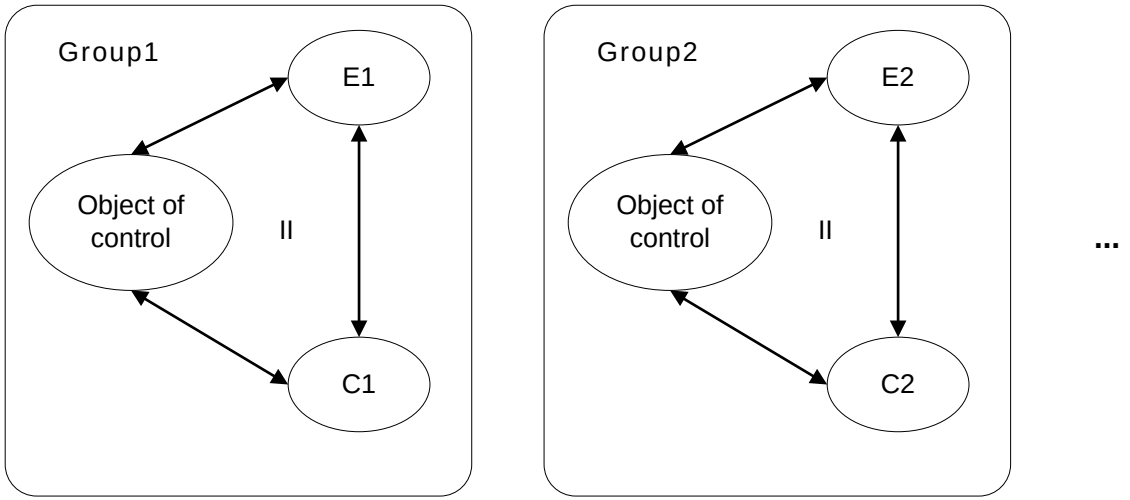
Minimal patterns

- Translate safety property into error states reachability (if no such property is specified, error states set is empty)

  A[] not (Err1 and Err2 and …)

- Translate liveness property into relation between good states (if no such property is specified, all good things we wish, will happen)





Group1 I= ψ1 Group2 I= ψ2 ...

Properties as ”error” states

- Ex: The model of interaction for the single-lane bridge consists of a set of CAR processes and a BRIDGE (environment) process that constrains the access of cars to the bridge. The problem is to ensure that cars moving in different directions cannot concurrently access the shared resource that is the bridge. To be more precise, the ***safety*** property we require of the bridge is that two cars must not be on the bridge at the same time, and the ***liveness*** property we require is that all cars eventually get to cross the bridge.

- An ERROR state, as the situation when two cars try to enter the bridge simultaneously, can be introduced, with ingoing transitions for all actions in the alphabet of the property. This means that a safety property can be composed with a set of processes without affecting their correct behaviour.
- The advantages of checking safety in this way are twofold:
  - Firstly, it is compositional in that property automata can be combined hierarchically with the components to which they apply.
  - Secondly, multiple properties can be checked at the same time. The check is implemented as a search for reachability of the ERROR state.

- In the bridge example, the liveness property means that any car process reaches the state of passing the bridge, or negation of the assertion, as a set of relations to an ERROR state.

COMDES verification lines related to MoDES project

- Generic compression algorithms for abstracting/translating COMDES sequence of states/transitions into patterns of state/transition in the formal verification model (Uppaal).
  - Correctness of a semantics mapping with state space reduction algorithms.
  - To implement iteratively reconfigurable forms of predicate abstraction as transformations in translation
- Pair-wise compositional proof of component-interaction automata
  - Property assertions, or negations of them, translated in automata/observers with error states
  - Sound decomposition of global properties into local
- Exploit causal and synchrony relations, like input-output chain-reaction sequences, or given in terms of behavioural type specifications, like A produces a and b at the same rate but B consumes two values of a for each value of b, and express them as annotations, to minimize the state-space explosion caused by nondeterministic interleaving of actions of the concurrent assembly of processes.