Model checking Hybrid Systems via Satisfiability Modulo Theories

Alessandro Cimatti
Embedded System Unit
Fondazione Bruno Kessler
Trento, Italy
cimatti@fbk.eu

Joint work with Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta

We gratefully acknowledge the support of the European Space Agency contracts OMC-ARE, COMPASS, IRONCAP, AUTOGEF, FOREVER, FAME, HASDEL.

Fondazione Bruno Kessler

- Fondazione Bruno Kessler
  - Private foundation with public finalities
  - Owned by Provincia Autonoma di Trento
  - Formerly IRST, Istituto Trentino di Cultura
- Center for Information Technology
  - Director: Paolo Traverso
- The Embedded Systems Unit
  - 28 people
  - 7 research staff, 7 postdocs, 8 programmers, 6 ph.d. students
  - Open call for more ph.d. students and postdocs!
- Strategy: tight integration of
  - Basic research
  - Tool development
  - Technology transfer

Take away messages

- The need for verification
  - Very complex systems
- Verification in a broader sense
  - Rigorous analysis of the behaviour of dynamic systems
- Hybrid automata
  - A uniform and comprehensive formal model
- Satisfiability Modulo Theories
  - Higher level symbolic modeling
  - Efficient engines: SAT + constraint solving
- SMT-based Verification
  - Many effective complementary algorithms
- Application in several project
  - Strong potential for practical impact

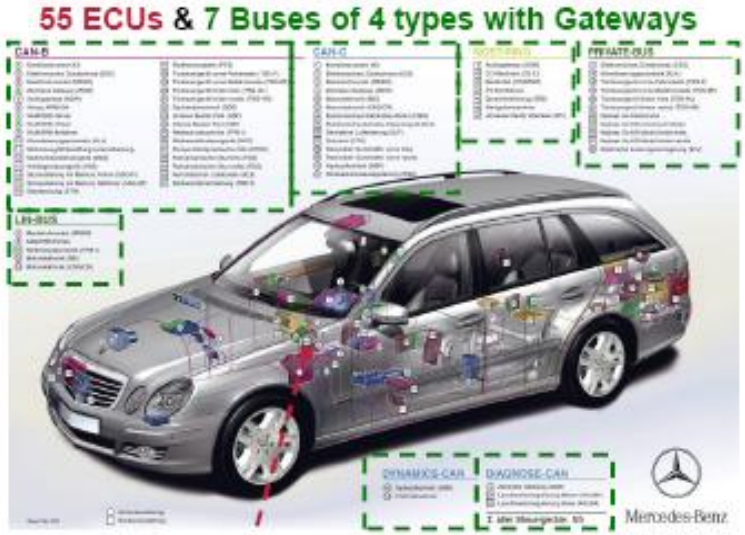
FONDAZIONE BRUNO KESSLER

Structure of the tutorial

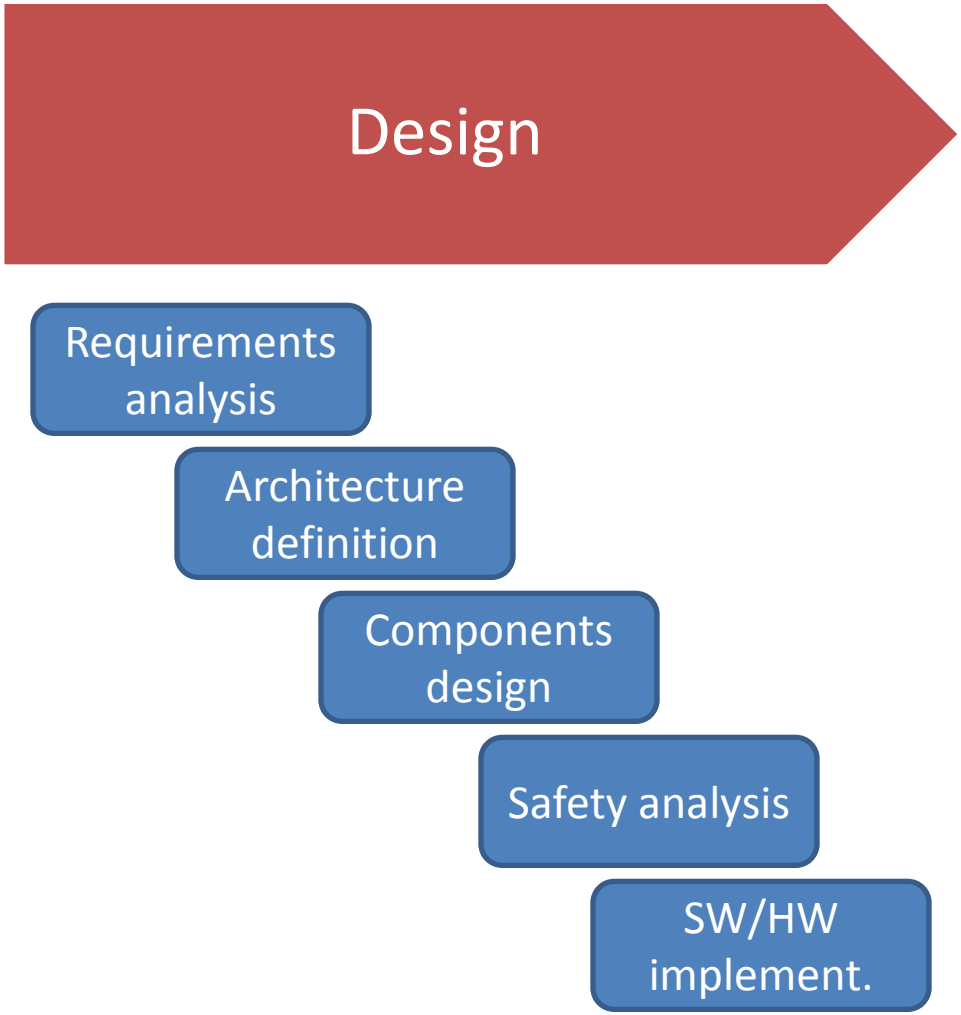
- Motivations
- Hybrid Systems
- Satisfiability Modulo Theories
- SMT-based verification
- SMT-based verification of Hybrid Systems
- Requirements analysis

The Design Challenge

- Designing complex systems
  - Automotive
  - Railways
  - Aerospace
  - Industrial production
- Sources of complexity:
  - Hundreds of functions
  - Networked control
  - Real-time constraints
  - Complex execution model with mixture of real-time and event-based triggers
  - System composed of multiple heterogeneous subsystems
  - Critical Functions:
    - » ABS, drive-by-wire
    - » Operate switches, level crossings, lights
    - » Manage on-board power production
  - Conflicting objectives:
    - » Avoid crashes vs move trains



Life Cycle of Complex Systems

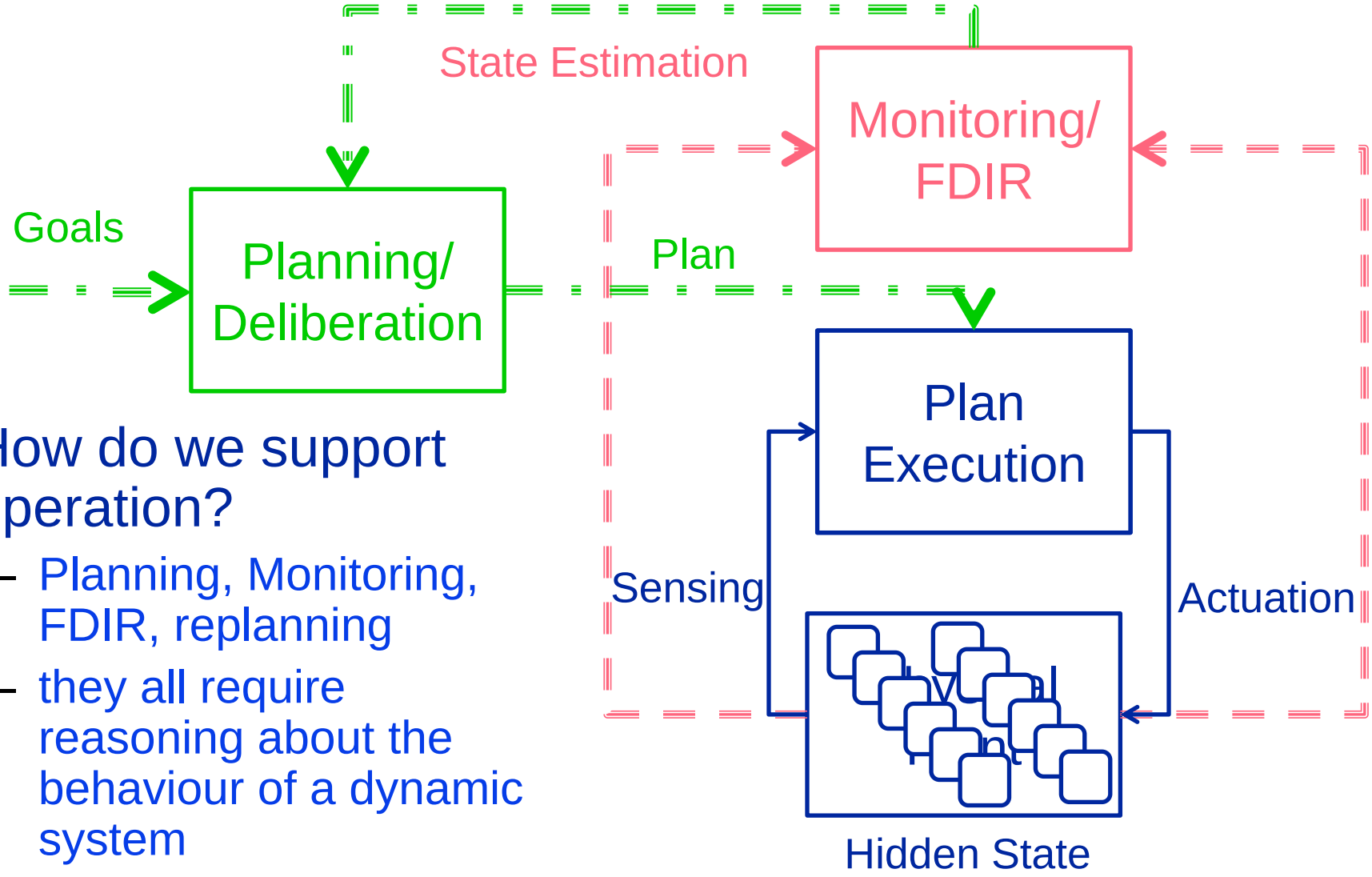


- How do we support the design?
- Requirements validation:
  - Are the requirements flawed?
- Functional correctness
  - Does the system satisfy the requirements?
- Safety assessment
  - Is the system able to deal with faults?

From design to operation…

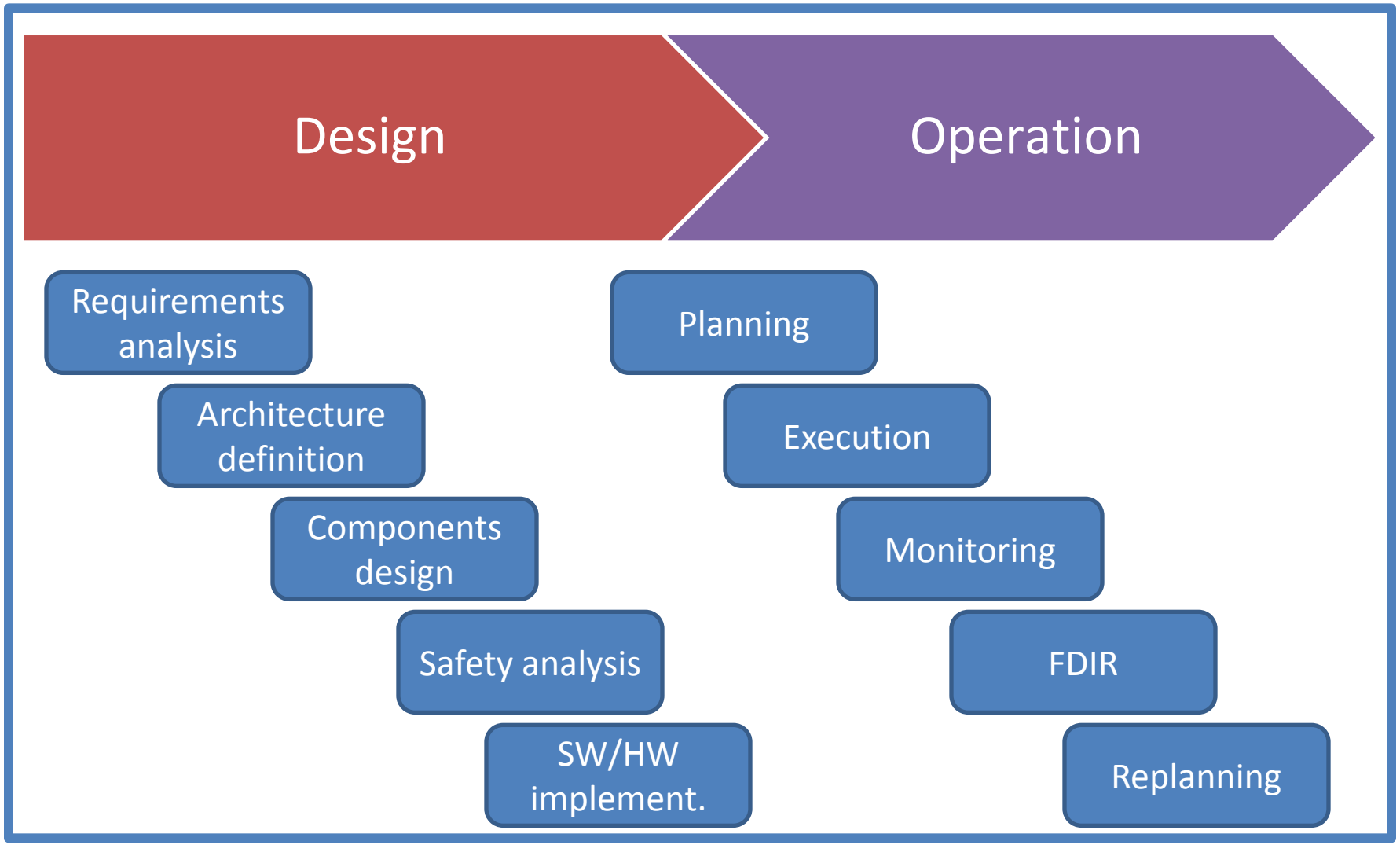
- Planning
  - plan how to achieve desired “firing” sequence
  - retrieve pipes from holds, pre-weld, send to firing line, final weld
- Execution Monitoring
  - welding may fail, activities can take more time than expected
  - plant may fail
- Fault Detection, Fault Identification/Isolation
  - is there a problem? where is it?
- Fault Recovery
  - put off-line problematic equipment
- Replanning
  - identify alternative course of actions, e.g. reroute pipes

Complex systems operation



- How do we support operation?
  - Planning, Monitoring, FDIR, replanning
  - they all require reasoning about the behaviour of a dynamic system

Life Cycle of Complex Systems



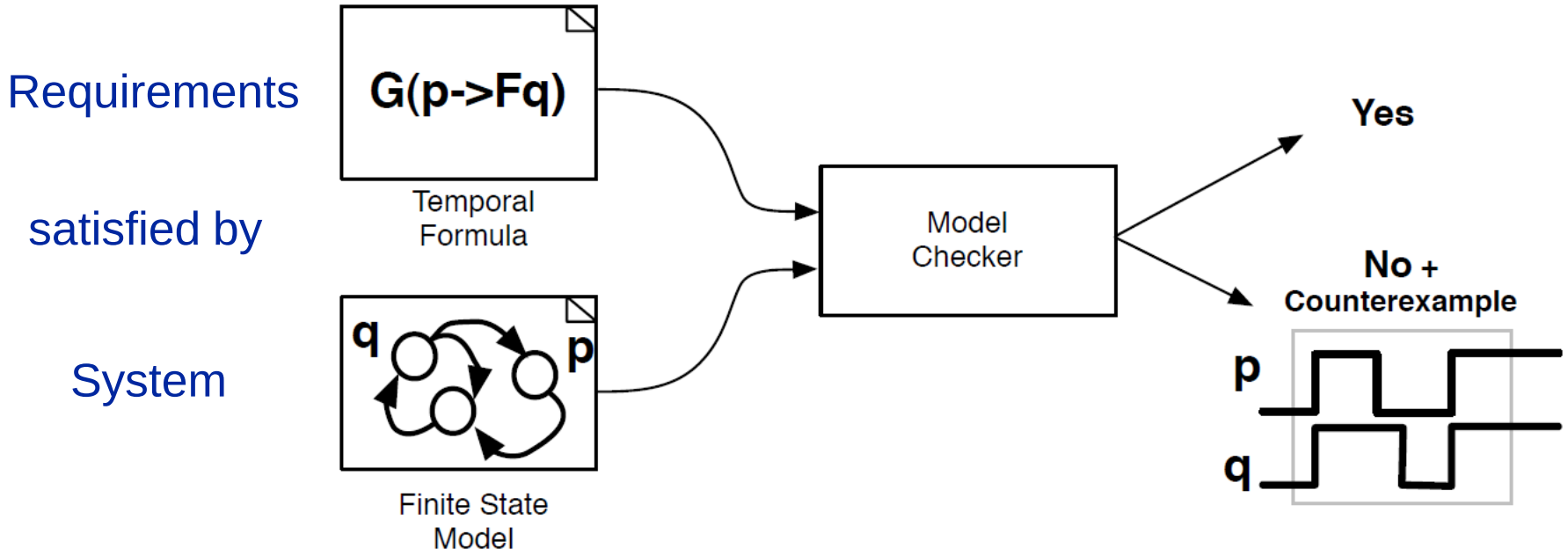
A formal approach

- Both design and operation tasks require
  - the analysis of the behaviour of dynamic systems over time
    - » In fact, they often require the analysis of the *same* dynamic systems
  - the analysis must be “rigorous”
    - » predictability, certification

- We need a rich formalism
  - to represent the behaviour of complex systems
  - to provide the reasoning tasks required for design and for operation

Model Checking in a nutshell

- Does system satisfy requirements?
- System as finite state model
- Requirements as temporal properties



Model checking

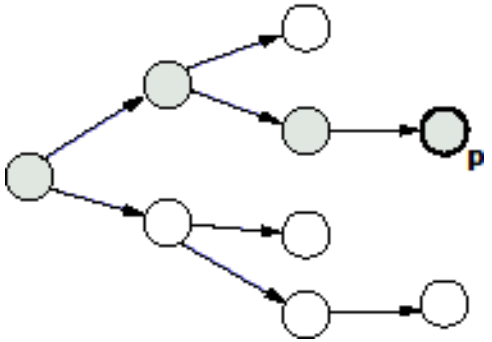
- Reactive System
  - infinite computation, interacting with environment
  - communication protocol, hw design, control software, OS
  - modeled as a (finite) state transition system
- Requirements
  - desirable properties of system behaviour
  - modeled as formulae in a temporal logic (CLT, LTL, PSL, …)
- Does my system satisfy the requirements?
  - Is the set of traces “generated” by the system included in the set of traces “accepted” by the requirements?
- Model checker
  - search configurations of state transition system
  - detect violation to property, and produce witness of violation
  - conclude absence of violation when fix point reached

Properties

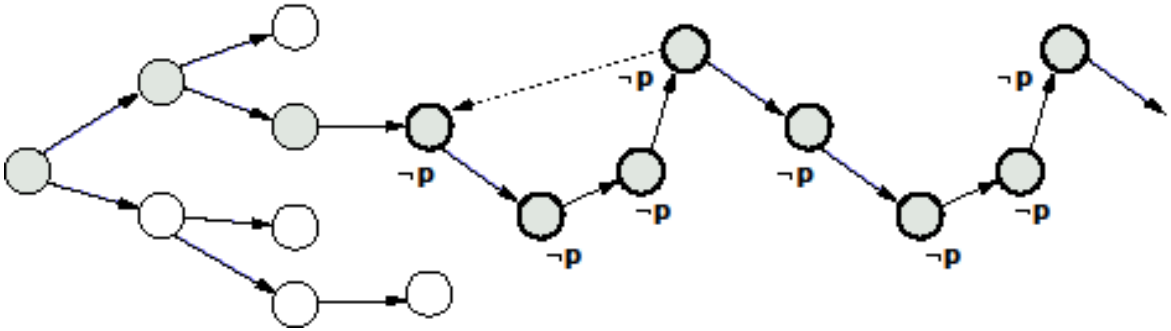
- Temporal logic can be used to express properties of reactive systems
- Safety properties: nothing bad ever happens
  - Two concurrent processes never execute simultaneously within their critical section
- Liveness properties: something desirable will eventually happen
  - A subroutine will eventually terminate execution and return control to the caller
  - Whenever a request arrives, it is sooner or later followed by a response

Refuting temporal properties

- Safety: refuted by finite trace to bad state



- Liveness: refuted by infinite trace with invariant suffix
  - Finitely presented as cycle



FONDAZIONE
BRUNO KESSLER

Modeling hybrid systems

Representation Challenges

- A formalism to characterize systems with
  - Nondeterministic behaviours
  - Possible faults
  - Operation in degraded modes
  - Limited observability
  - Parallel actions/tasks
    - » Start actuations in different subsystems
  - Activities with duration
    - » Time taken by procedures
    - » e.g. moving, welding, checking, …
  - Resources
    - » Power consumption, space, bandwidth, memory, …

Modeling (I)

- Synchronous, finite case
  - Circuits
- Finite state
  - each state variable associated with value in finite range

```
VAR x, y: boolean
init(x) := 0, init(y) := 0
next(x) := !x
next(y) := if x then !y else y
```

- Synchronous composition
  - Both variables evolve at the same time

```
x: 0 1 0 1 0 1 0 1 ...
y: 0 0 1 1 0 0 1 1 ...
```

Modeling (II): infinite data

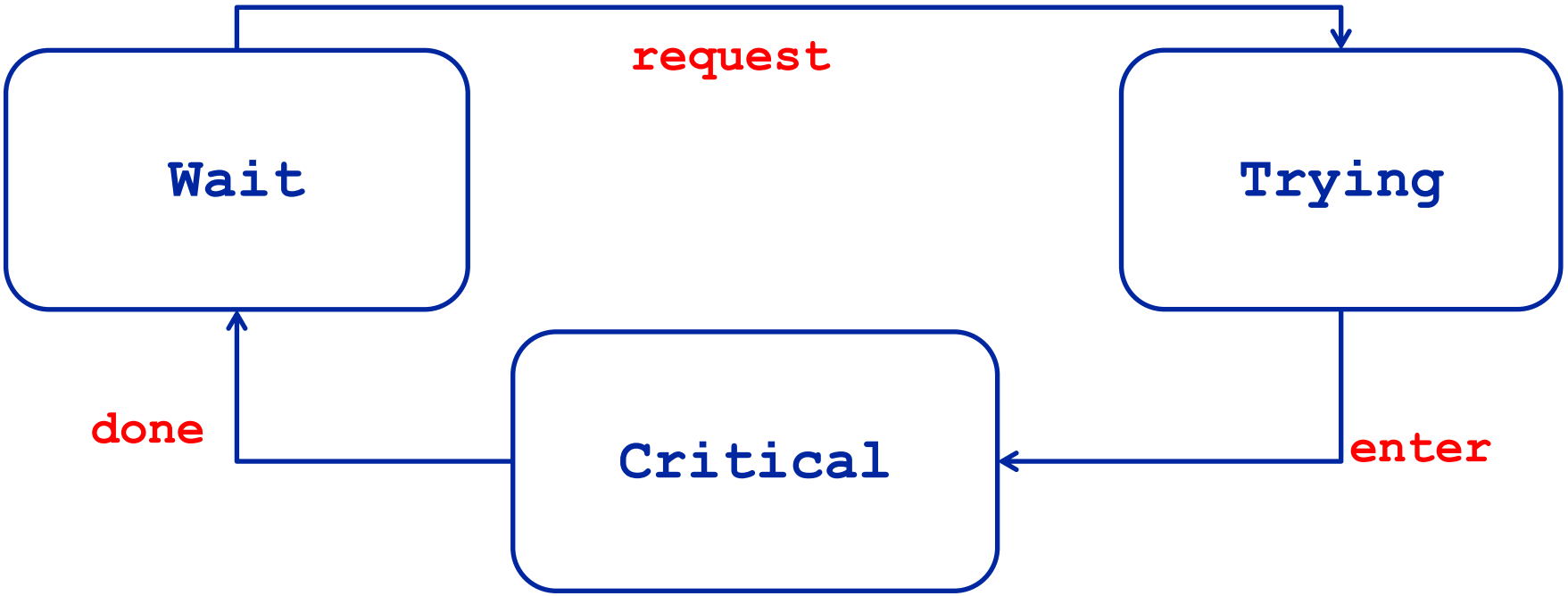
- Synchronous, infinite case
  - programs
- Infinite state: each state variable associated with value in finite range

```
VAR n : integer;
next(n) := if (even n)
             then (n / 2)
             else (3*n + 1)
```

- Reaching a fix point no longer guaranteed

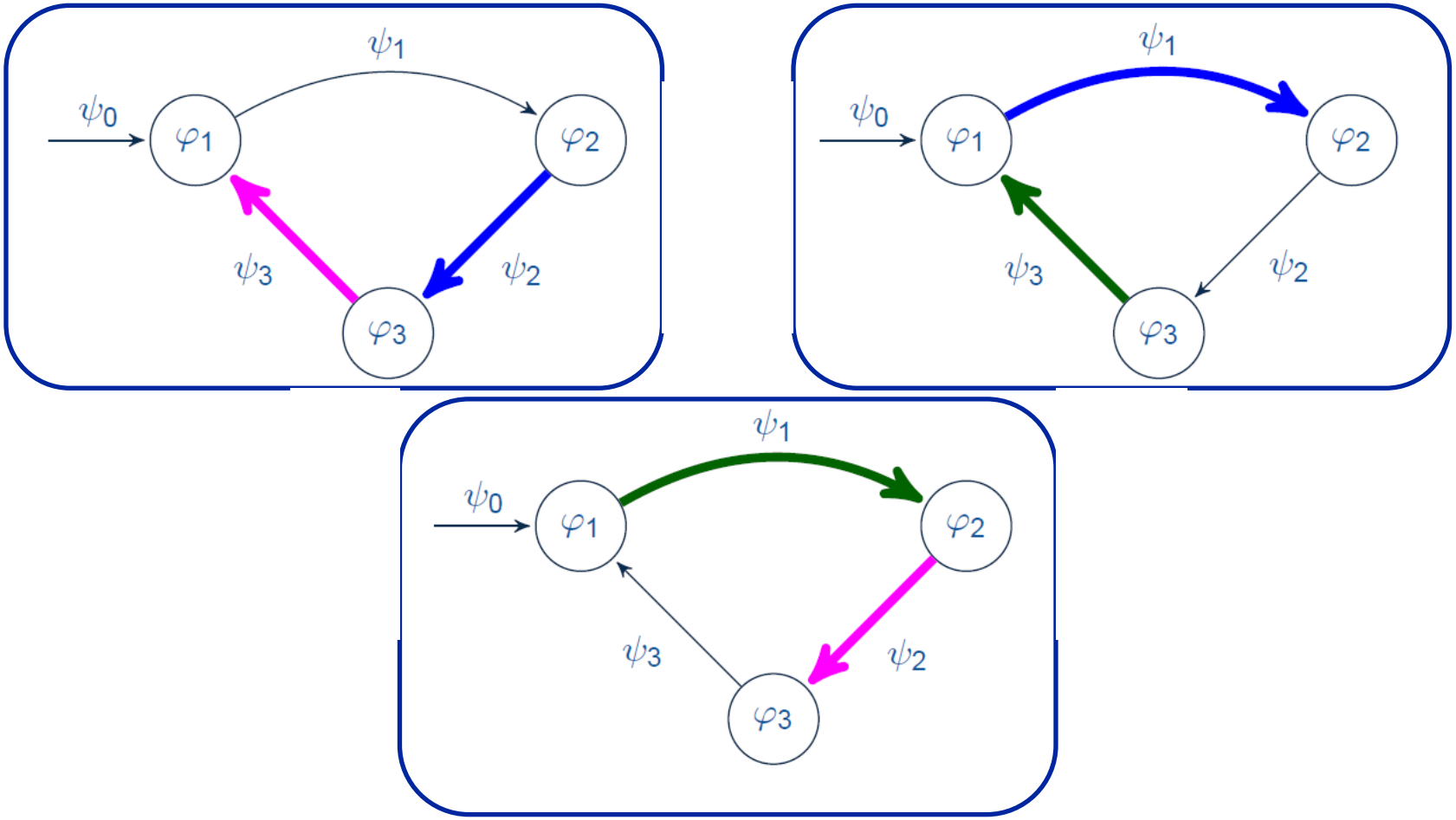
Modeling (III): asynchronous composition

- Automaton with states and transitions




```
VAR s : { Wait, Trying, Critical};
IVAR label : { req, enter, done, stutter};
s=Wait & label = request -> next(s)=Trying
label = stutter -> next(s)=s
```

Modeling (III): Networks of automata




```
SYNC server.grant1 C1.enter
SYNC server.grant2 C2.enter
...
```

Symbolic Representation

- State variables as variables in a logical language
  - x, y, z, w
- A state is an assignment to state variables
  - The bitvector 0011
  - The assignment { z, w }
  - The formula ¬x ∧ ¬y ∧ z ∧ w
- A set of states is a set of assignments
  - can be represented by a logical formula
  - x ∧ ¬y represents {1000, 1001, 1010, 1011} or a larger set, if more variables are present
- Set operations represented by logical operations
  - union, intersection, complementation as disjunction, conjunction, negation
- I(X), B(X) are formulae in X
  - Is there a bad initial state?
  - Is I(X) ∧ B(X) satisfiable?

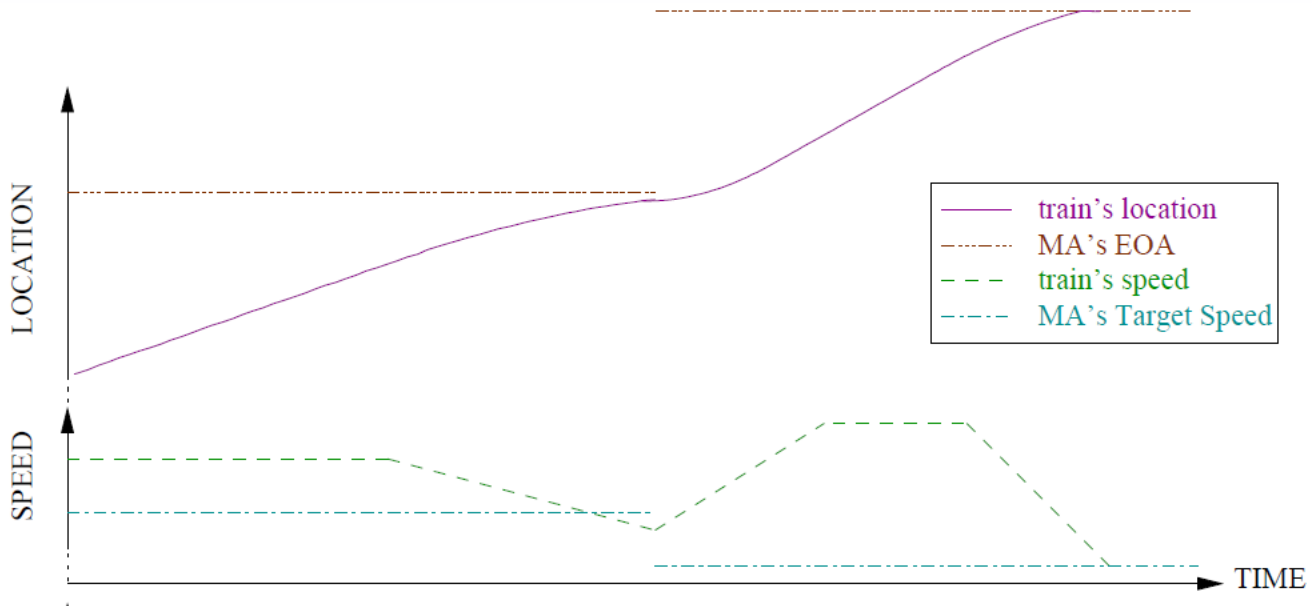
Symbolic Representation

- Symbolic representation not only for finite case!
  - Software: control flow graph + data path
  - Hardware at RTL, SystemC, threaded software
  - UML state machines, AADL descriptions
- Transition
  - pair of assignments to state variables
- Use two sets of variables
  - current state variables: x, y, z
  - next state variables: x', y', z'
- A formula in current and next state variables
  - represents a set of assignments to X and X'
  - a set of transitions
  - R(X, X')

From discrete traces to hybrid traces

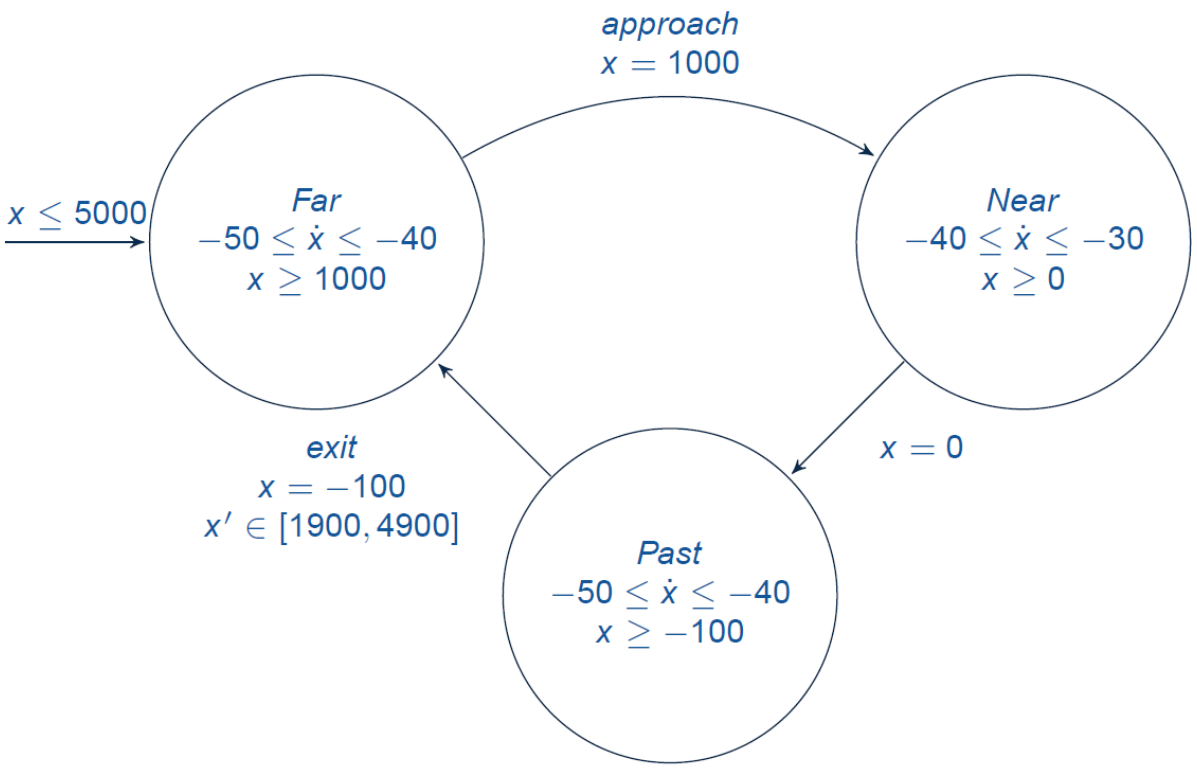
- So far
  - no notion of real time
  - traces as sequences of assignments to state variables
- This is often not enough
- Example:
  - Train moving on track
  - Evolution of position and speed over time
  - Movement authorithy (MA):
    - » Proceed until position “end of authority” (EOA)
    - » At EOA speed must be below “target speed” (TS)

Hybrid means discrete + continuous



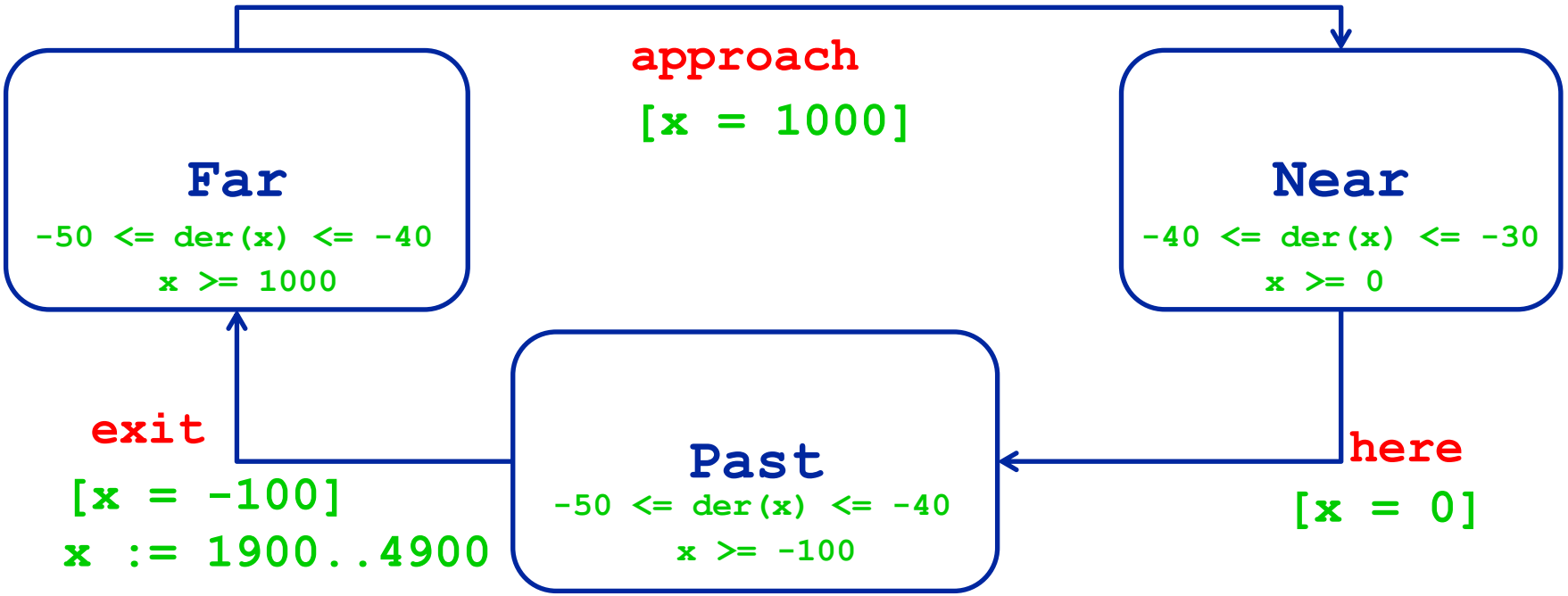
- State as values to variables
  - discrete variables
    - » Operation modes
  - continuous variables
    - » Speed, position
- Transitions from state to state
- Continuous transitions
  - Discrete component does not change
  - time elapses
  - Continuous variables evolve accordingly
- Discrete transitions
  - Instananeous
  - Discrete component changes
  - Continuous component may have jumps
    - » Timer reset
    - » Speed limit variation

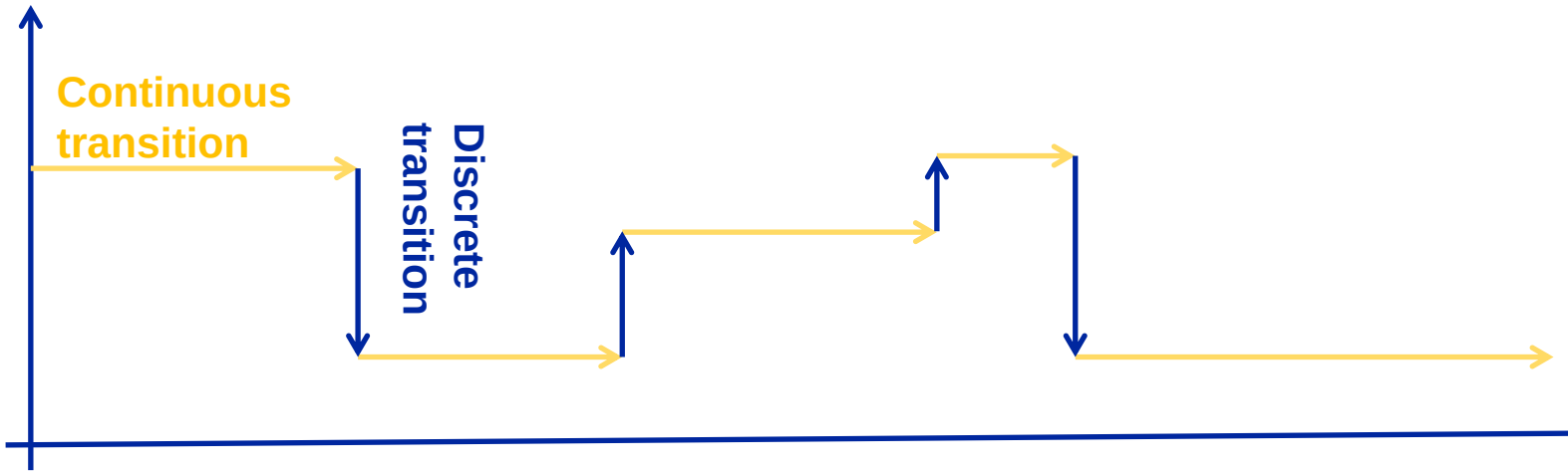
The formalism: hybrid automata



- Locations
- Events
- Transitions
- Continuous variables
- Guards
  - Enable transtions
- Invariants
  - Must be satisfied in locations
- Flow conditions
  - How do variables evolve when time elapses

Hybrid automata





Properties of hybrid automata

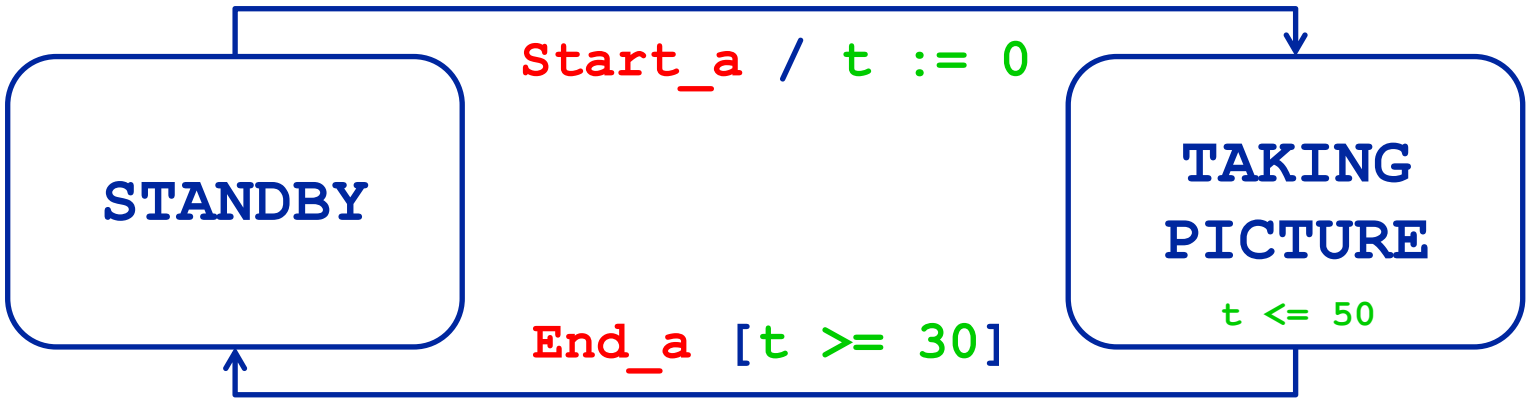
- Well founded, comprehensive and well studied
  - Clear definition of behaviors of model
  - Which states are reachable
- Temporal properties to express scenarios and requirements
  - *never two processes in critical region*
  - *always if req then within 5 sec response*

- Model checking
  - » Does the system satisfy the requirements?
- Temporal reasoning
  - » Strong/weak/dynamical controllability?
- Planning
  - » Find the inputs that will bring the system to required state

- The workhorse: satisfiability modulo theories

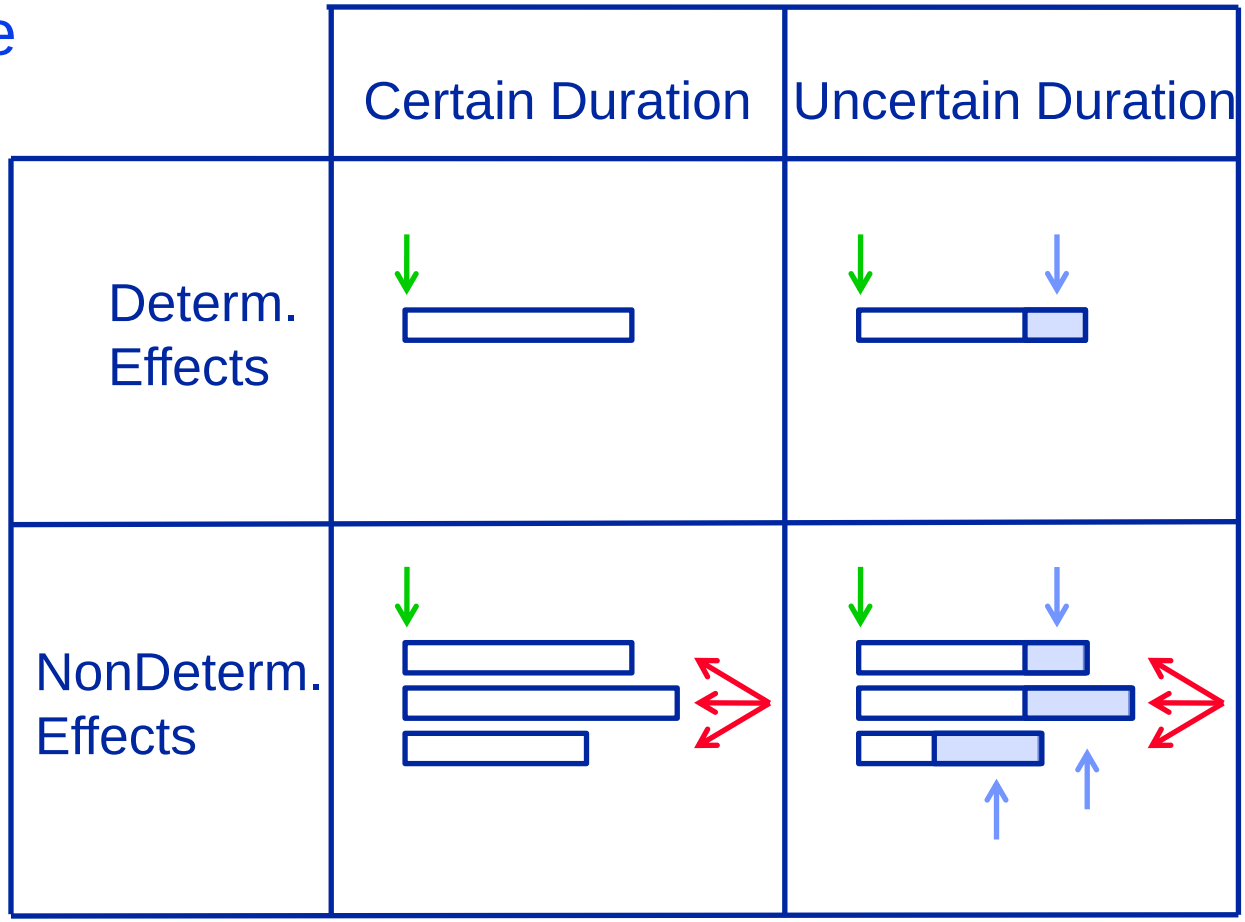
An example

```
Start_a -> s = STANDBY
Start_a -> next(s) = TAKING_PICTURE
Start_a -> next(t) = 0.0

s = TAKING_PICTURE -> t <= 50.0

End_a -> s = TAKING_PICTURE
End_a -> next(s) = STANDBY
End_a -> t >= 30.0
```

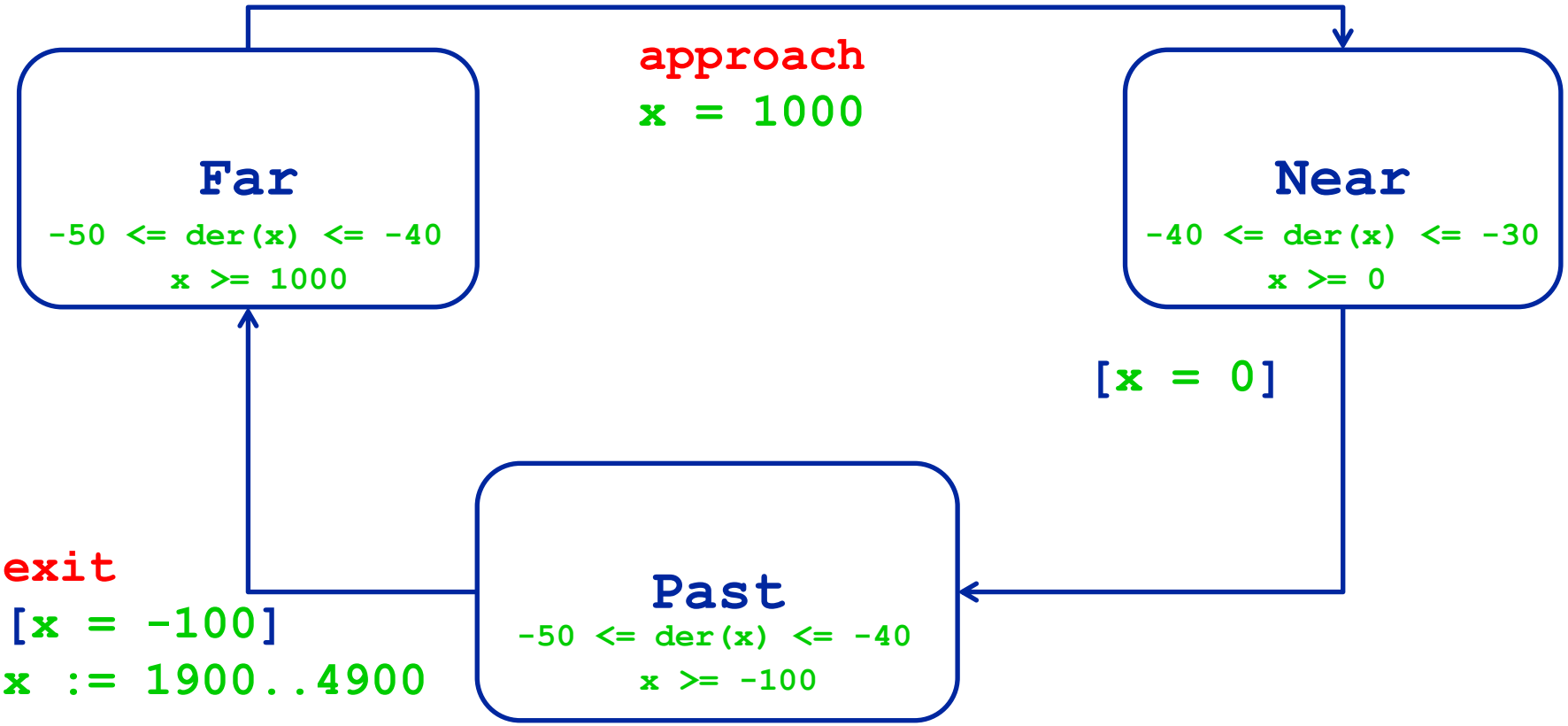


Nondeterminism and uncertainty

- Nondeterminism
  - Discrete choice
- Uncertainty
  - Continuous
- Controllable
  - Start
- Uncontrollable
  - Effects
  - End



From HA to SMT formulae




```
s = Past -> x >= -100
exit -> s = Past & x = -100
exit -> next(s) = Far
exit -> next(x) in 1900..4900
```

```
timed -> next(s) = s
timed & s = Past ->
    next(x) >= x - 50*delta &
    next(x) <= x - 40*delta
```

The SMT representation

```
VAR s : { Past, Near, Far }
VAR x : real;
...
INIT x <= 5000
INIT s = Past
...
TRANS
s = Past -> x >= -100
exit -> s = Past
exit -> next(s) = Far
exit -> next(x) >= 1900
exit -> next(x) <= 4900
...
timed -> next(s) = s
timed -> next(x) >= x - 50*delta
timed -> next(x) <= x - 40*delta
```

Hybrid automata symbolically represented by SMT formulae!

I(X) initial states
R(X,X') transition relation
B(X) bad/target states

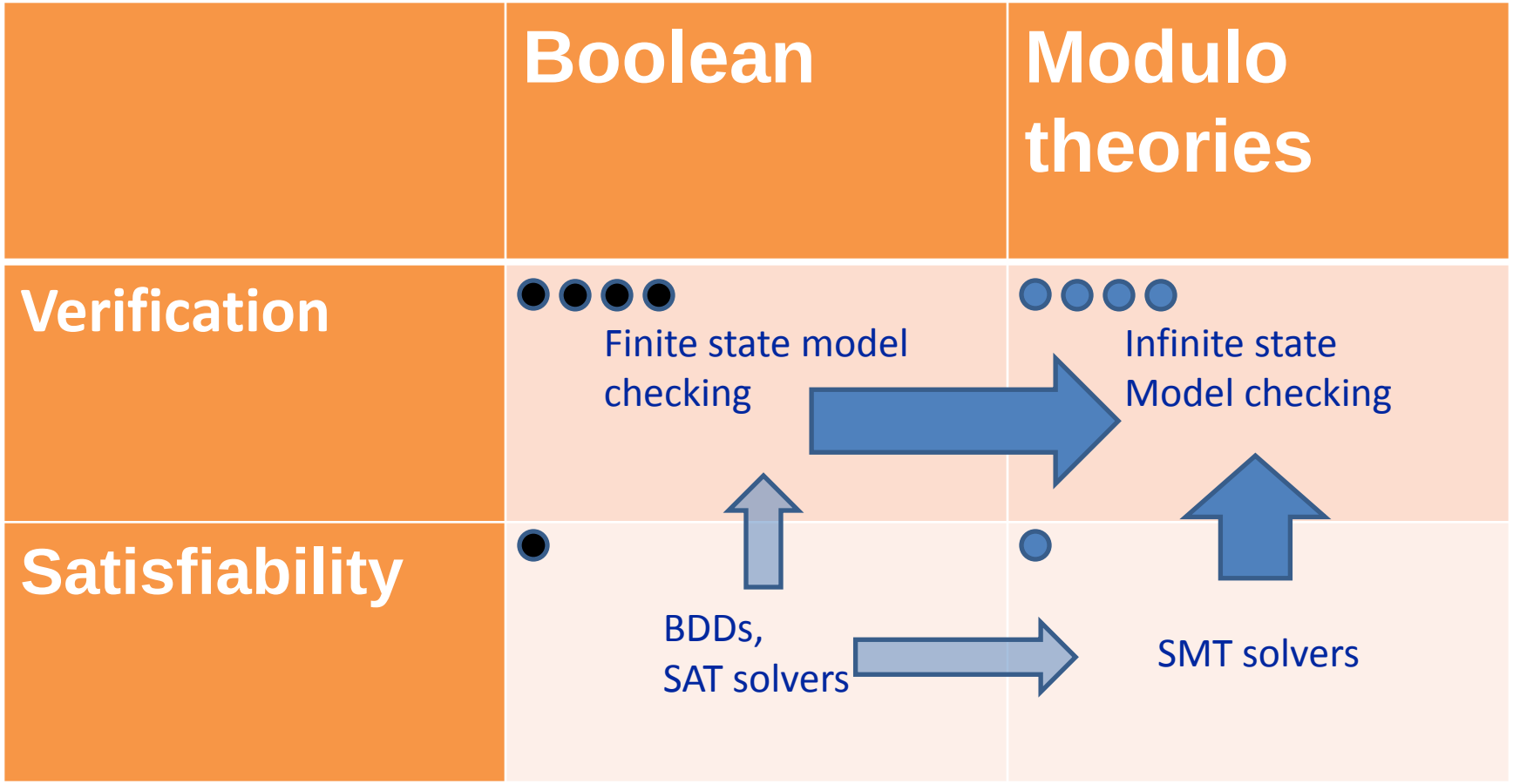
FONDAZIONE
BRUNO KESSLER

Engines for symbolic verification

From SAT to SMT

Satisfiability vs Verification

(or, combinational vs sequential)



Underlying engines

- Finite case
  - Binary Decision Diagrams
  - Boolean Satisfiability Solving
- Infinite case
  - Satisfiability Modulo Theories

Binary Decision Diagrams

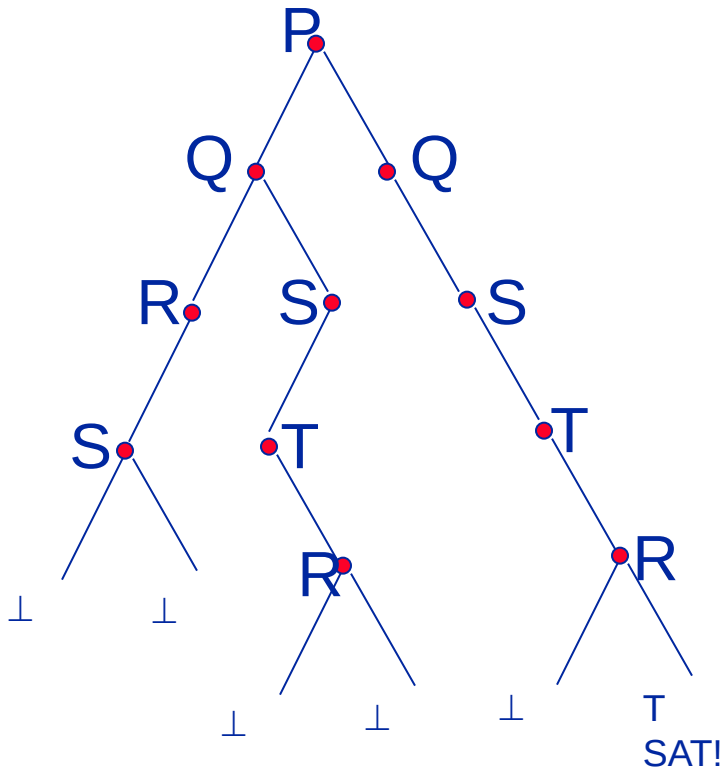
- Representation of boolean functions
- Canonical form for propositional logic
- Widely used in formal verification
- Efficient BDD packages provide
  - boolean operations
  - universal and existential quantification (QBF)
  - caching and memoizing

- Used to represent
  - accumulated states
  - partial policies

BDD-based Symbolic Model Checking

- Based on Binary Decision Diagrams
  - canonical representation for logical formulae
  - boolean operations, quantifier elimination
- I(X), R(X, X'), B(X)
  - each represented by a BDD
- Image computation: compute all successors of all states in S(X)
  - based on projection operation
  - exists X.(S(X) and R(X, X'))
- Reachability algorithm
  - Expand new states until bug, or fix point

Boolean DPLL



- The DPLL procedure
- Incremental construction of satisfying assignment
- Backtrack/backjump on conflict
- Learn reason for conflict
- Splitting heuristics

Satisfiability modulo theories

- Satisfiability of a first order formula …
  - where the atoms are interpreted modulo a background theory
- Theories of practical interest
  - Equality Uninterpreted Functions (EUF)
    - » $x = f(y), h(x) = g(y)$
  - Difference constraints (DL)
    - » $x – y \leq 3$
  - Linear Arithmetic
    - » $3x – 5y + 7z \leq 1$
    - » reals (LRA), integers (LIA)
  - Arrays (Ar)
    - » read(write(A, i, v), j)
  - Bit Vectors (BV)
  - Their combination

Statisfiability Modulo Theories

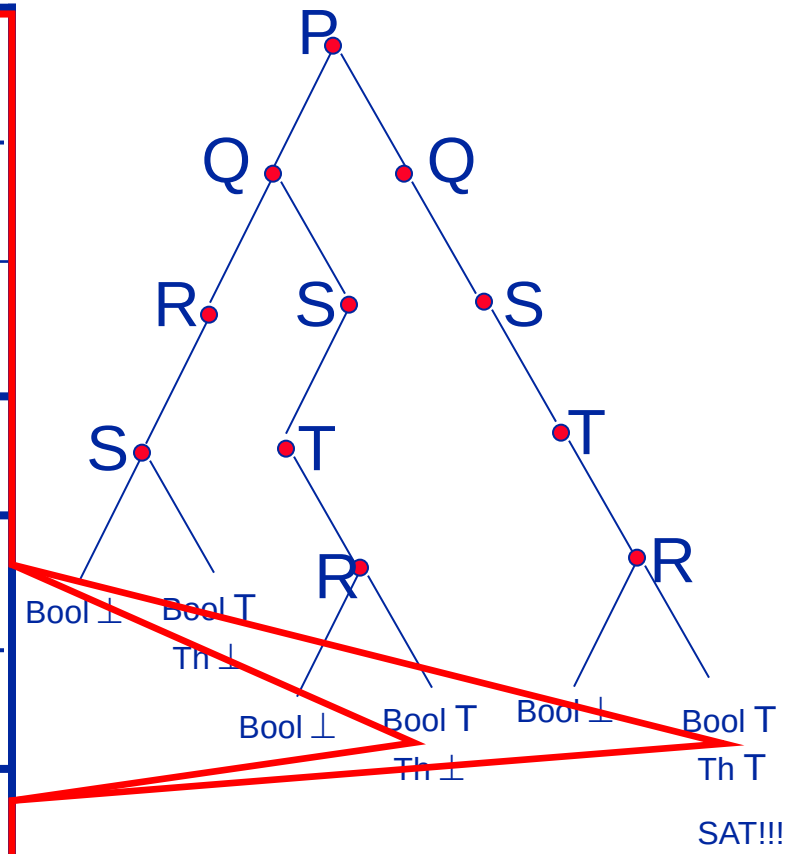
- An extension of boolean SAT
- Some atoms have non-boolean (theory) content
  - » A1 : $x - y \leq 3$
  - » A2 : $y - z = 10$
  - » A3 : $x - z \geq 15$
- Theory interpretation for individual variables, constants, functions and predicates
  - » if $x = 0, y = 20, z = 10$
  - » then A1 = T, A2 = T, A3 = F
- Interpretations of atoms are constrained
  - » A1, A2 and A3 can not be all true at the same time

SMT solvers

- Boolean reasoning + constraint solving
  - SAT solver for boolean reasoning
  - theory solvers to interpret numerical constraints

MathSAT: search space

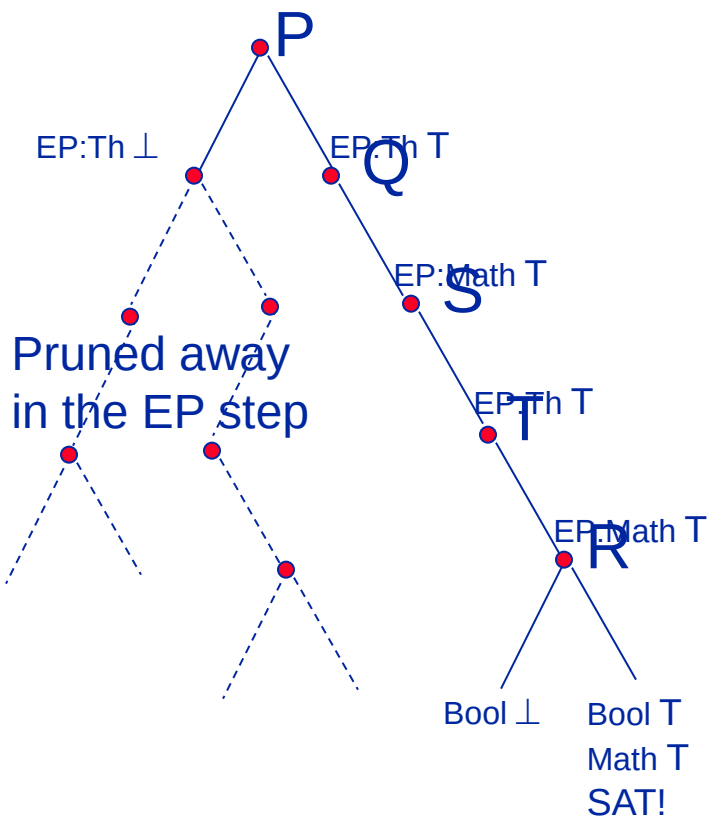
P	T	$x - y \leq 3$
P_1	F	
P_2	T	$y - z = 10$
Q	F	
R	T	$x - z \geq 15$
R_1	F	
S	F	$z - 2*w = 1$
S_1	T	



Many boolean models are not theory consistent!

Early pruning

Check theory consistency of partial assignments



Learning Theory Conflicts

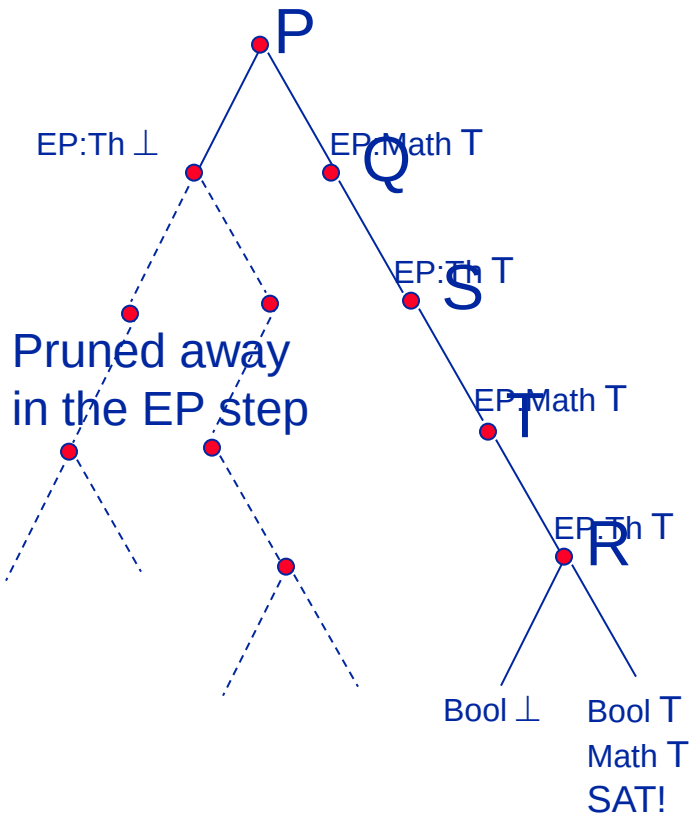
The theory solver can detect a reason for inconsistency

I.e. a subset of the literals that are mutually unsatisfiable

E.g. x = y, y = z, x != z

Learn a conflict clause
x != y or y != z or x = z

By BCP the boolean enumeration will never make same mistake again



Theory Deduction

The theory solver can detect that certain atoms have forced values

E.g. from x = y and x = z
infer that y = z should be true

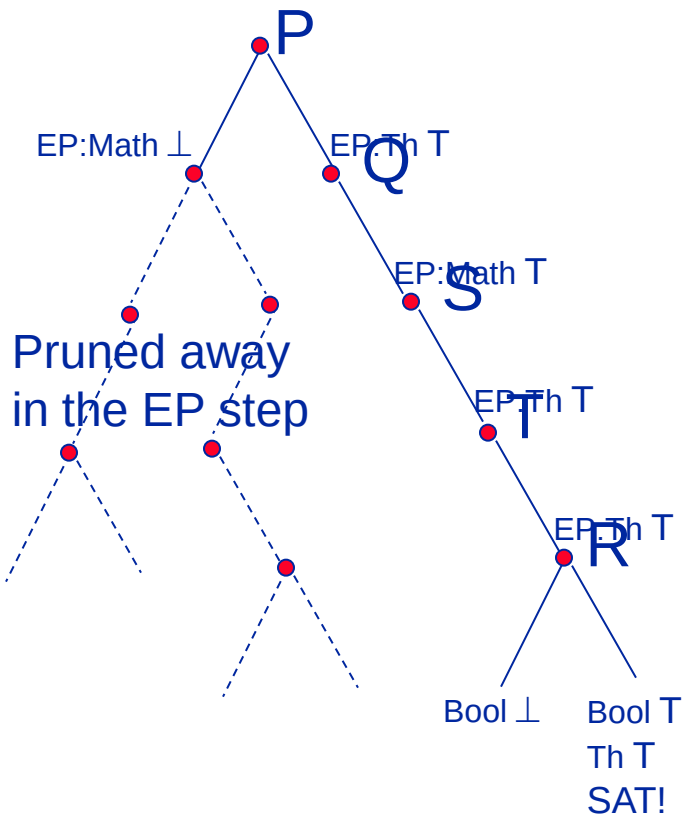
Force deterministic assignments

Theory version of BCP

Furthermore, we can learn the deduction:

x=y & x = z -> y=z

Theory Conflict vs theory deduction



Optimizations

- Incrementality and Backtackability
  - add constraints without restarting from scratch
  - remove constraints without paying too much
- Limiting cost of early pruning
  - filtering, incomplete calls
- Conflict set minimization
  - return T-inconsistent subset of assignment
- Deduction
  - return forced values to unassigned theory atoms
- Static learning
  - precompile obvious theory reasoning reasoning to boolean

SMT solvers in practice

- In practice, the integration is very tight
  - SAT solver working as an enumerator
  - Theory solver follows the stack-based search
    - » Inconsistent partial assignments are pruned on the fly
    - » conflicts clauses learnt from theory reasoning
    - » used to drive search at the boolean level

- Additional features
  - Model construction
  - Incremental interface
  - Unsatisfiable core
  - Proof production
  - Interpolation

- Satisfiability Modulo Theories: a sweet spot?
  - increase expressiveness
  - retain efficiency of boolean reasoning
- Trade off between expressiveness and reasoning
  - SAT solvers: boolean case, automated and very efficient
  - theorem provers: general FOL, limited automation

The SMT community

- Standard language and benchmarks
  - **http://www.smt-lib.org**
- Yearly competition
  - **http://www.smt-comp.org**
- Solvers
  - YICES, OpenSMT, Z3, CVC, …
- The MathSAT solver
  - **http://mathsat.fbk.eu**
  - Solving, core extraction, interpolation, allsmt, costs

Notable achievements

- Successful applications in various fields
  - verification of pipelined microprocessors
  - equivalence checking of Microcode
  - software verification
  - whitebox testing for security applications
  - design space exploration, configuration synthesis
  - discovery of combinatorial materials

- Reasons for success?
  - allows to deal with richer representation
  - increase capacity by working above the boolean level

FONDAZIONE
BRUNO KESSLER

SMT-based verification

Symbolic Encoding

- Vectors of state variables
  - current state X
  - next state X'
- Initial condition I(X)
- Transition relation R(X, X')
- Bug states B(X)

- Key difference
  - X, X' are not limited to boolean variables
    - » in addition to discrete
    - » reals, integers, bitvectors, arrays, …
  - I, R, B are SMT formulae

- Representation
  - higher level
  - structural information is retained

Bounded Model Checking

- State variables replicated K times
  - X_0 , X_1, …, X_{k-1}, X_k
- Look for bugs of increasing length
  - $I(X_0) \wedge R(X_0, X_1) \wedge \ldots \wedge R(X_{k-1}, X_k) \wedge B(X_k)$
  - bug if satisfiable
  - increase k until …
- Advanced use of satisfiability solver
  - incremental interface
  - theory lemmas should be retained
  - theory lemmas can be shifted over time
    - » from $\Phi(X_0, X_1)$ to $\Phi(X_i, X_{j+1})$
  - Unsat core and generation of interpolants
  - Elimination of quantifiers

Induction

- Prove absence of bugs by induction

$I(X_0) \wedge B(X_0)$
$\neg B(X_0) \wedge R(X_0, X_1) \wedge B(X_1)$
. . .
$I(X_0) \wedge R(X_0, X_1) \wedge \ldots \wedge R(X_{k-1}, X_k) \wedge B(X_k)$
$\neg B(X_0) \wedge R(X_0, X_1) \wedge \ldots \wedge \neg B(X_{k-1}) \wedge R(X_{k-1}, X_k) \wedge B(X_k)$

- Proved correct if unsatisfiable (and no bugs until *k)*
- Commonly used techniques
  - Invariant strengthening
    - » Sometimes trying to prove a stronger fact may be easier
  - Simple path condition
    - » Explore only paths that do not contain repetitions

Interpolation

An interpolant for an unsatisfiable formula

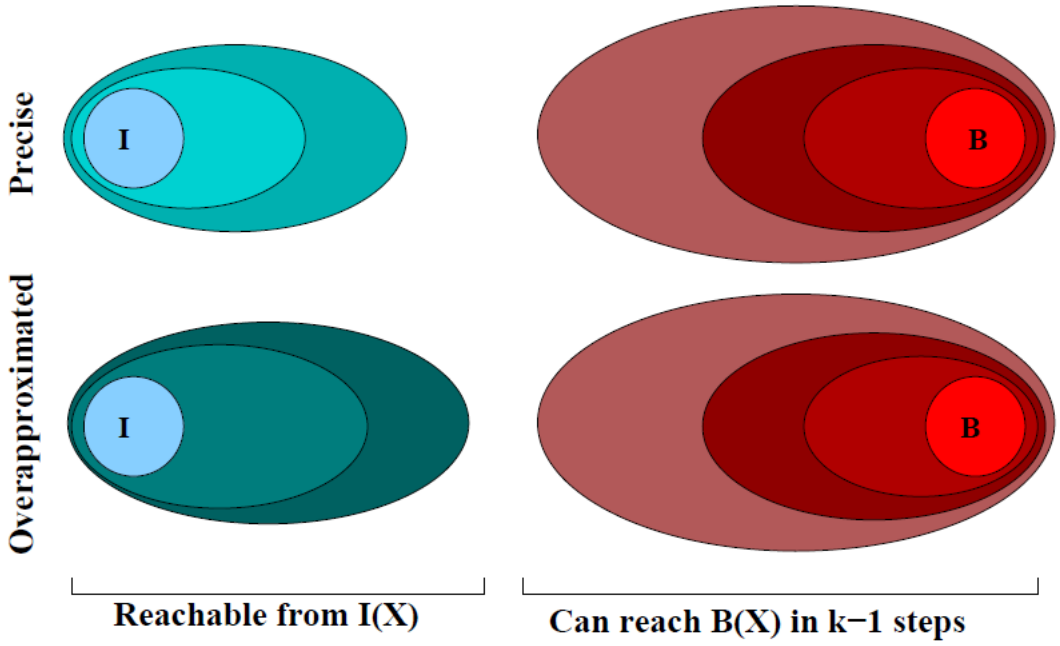
$$\Phi_1(X, Y) \wedge \Phi_2(Y, Z)$$

is a formula $Itp(Y)$ such that:

- $\Phi_1(X, Y) \rightarrow Itp(Y)$
- $Itp(Y) \wedge \Phi_2(Y, Z)$ is unsatisfiable

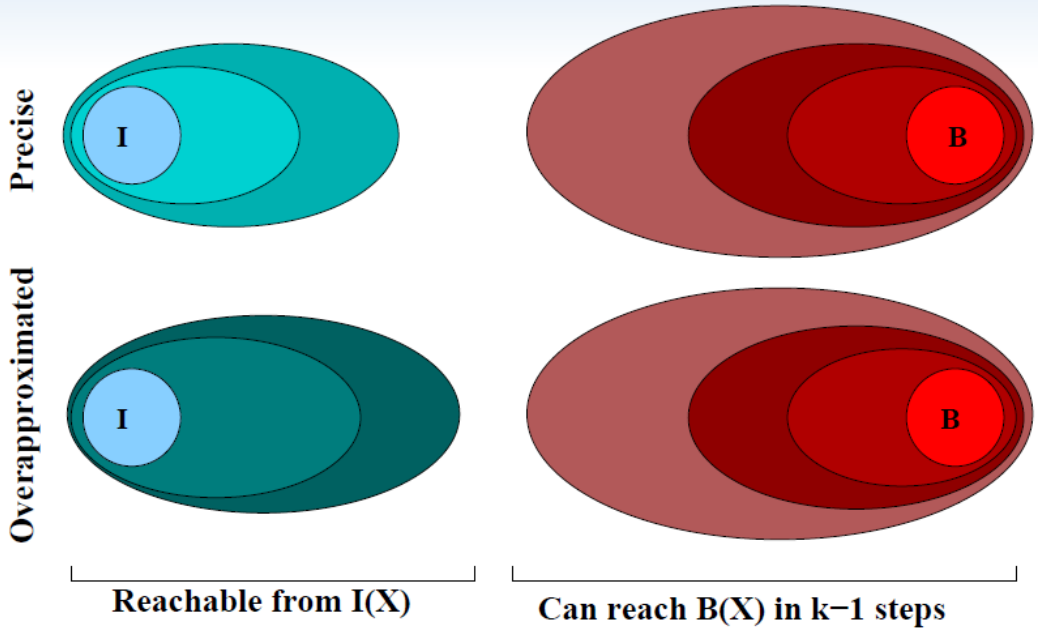
Interpolation-based model checking

$$\overbrace{I(X_0) \wedge R(X_0, X_1)}^{\Phi_1(X_0,X_1)} \underbrace{\wedge}_{Itp(X_1)} \overbrace{R(X_1, X_2) \ldots \wedge R(X_{k-1}, X_k) \wedge B(X_k)}^{\Phi_2(X_1,\ldots,X_k)}$$



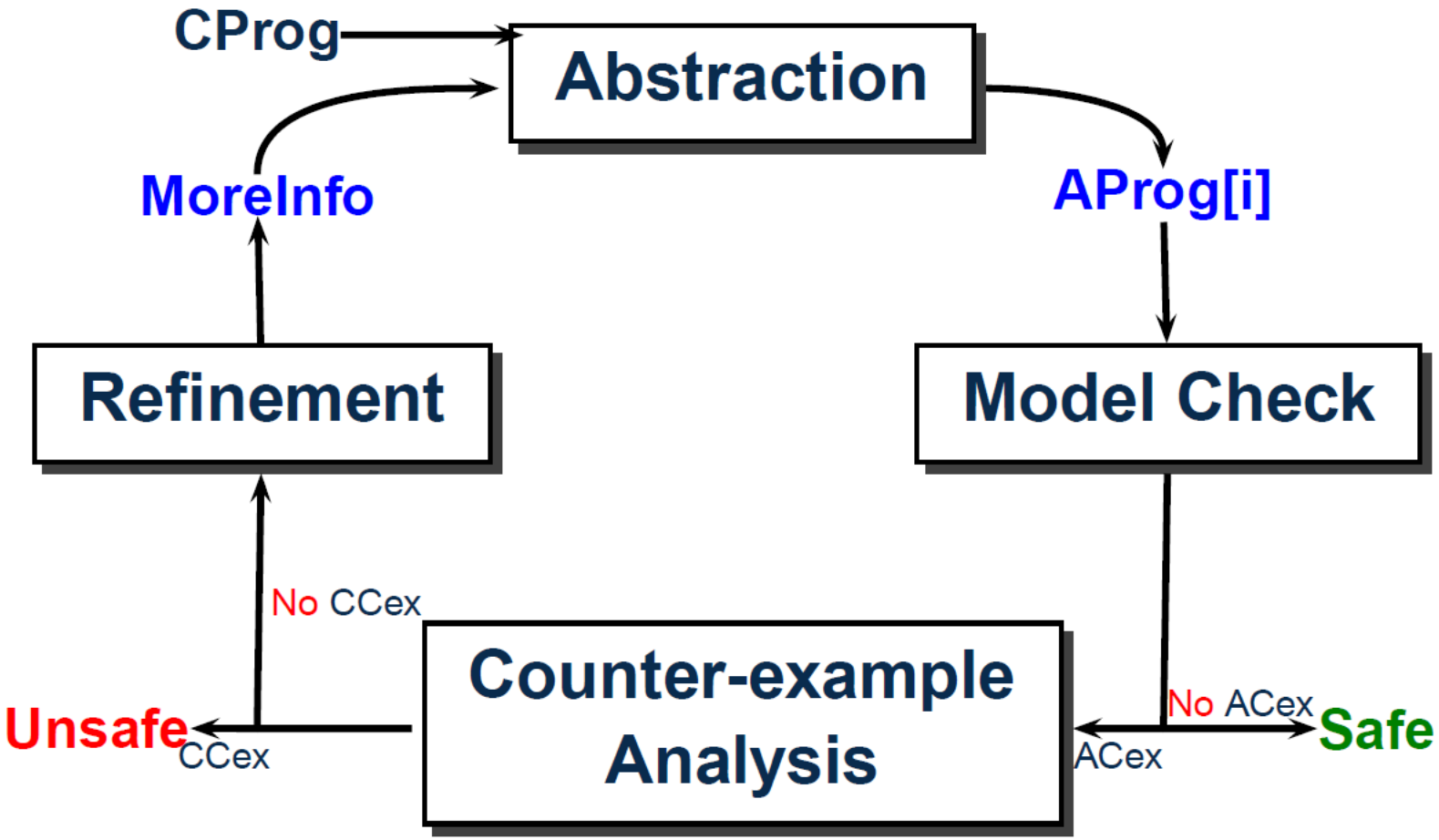
$$Itp(X_1) = Itp(R, I(X_0), k)$$

Interpolation-based model checking

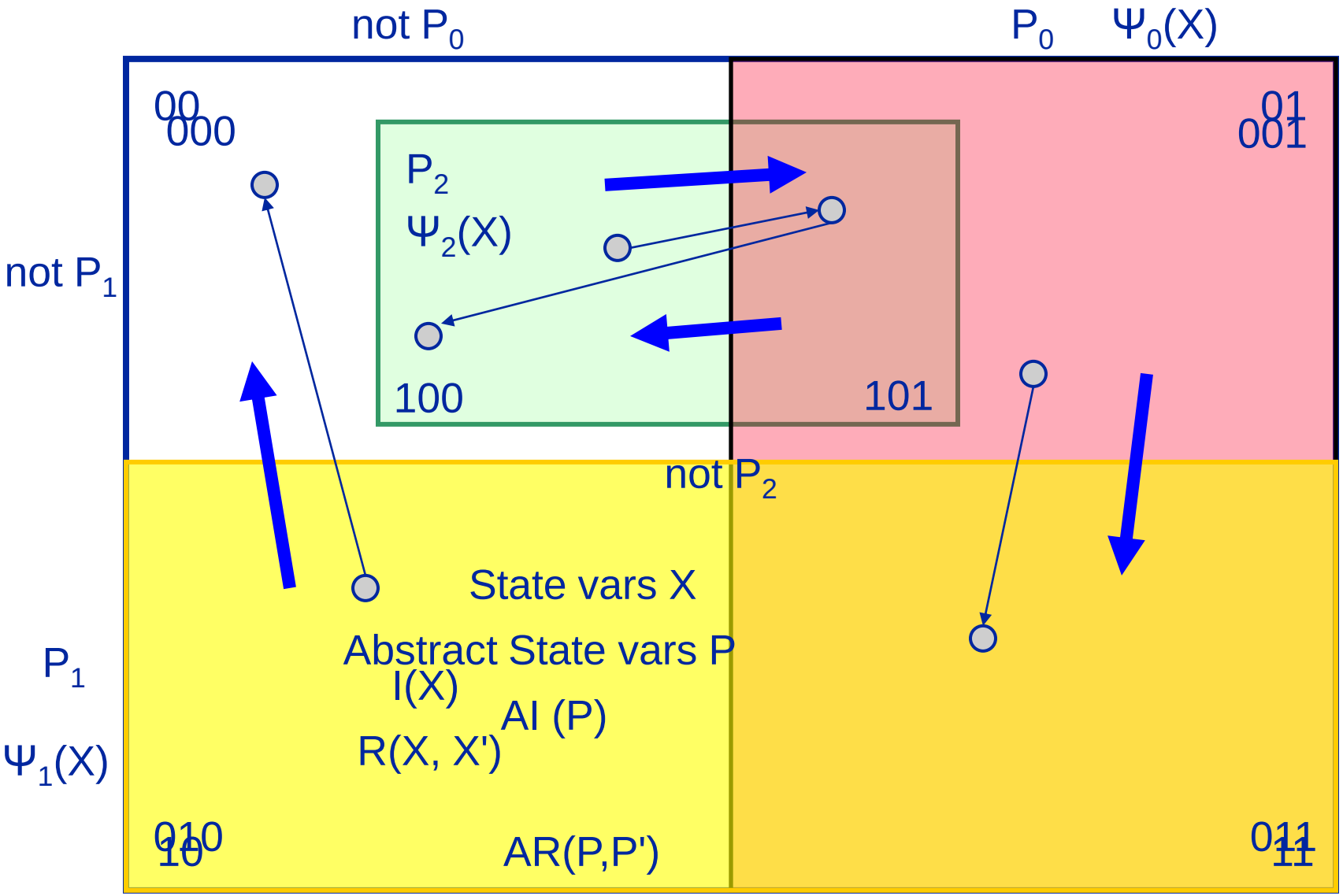


- Precise reachability
  - $\mathcal{R}_0 = I$
  - $\mathcal{R}_i = Img(R, \mathcal{R}_{i-1}) \cup \mathcal{R}_{i-1}$
- Interpolation based reachability
  - $Itp_0 = I(X_1)$
  - $Itp_i = Itp(R, Itp_{i-1}, k) \cup Itp_{i-1}$

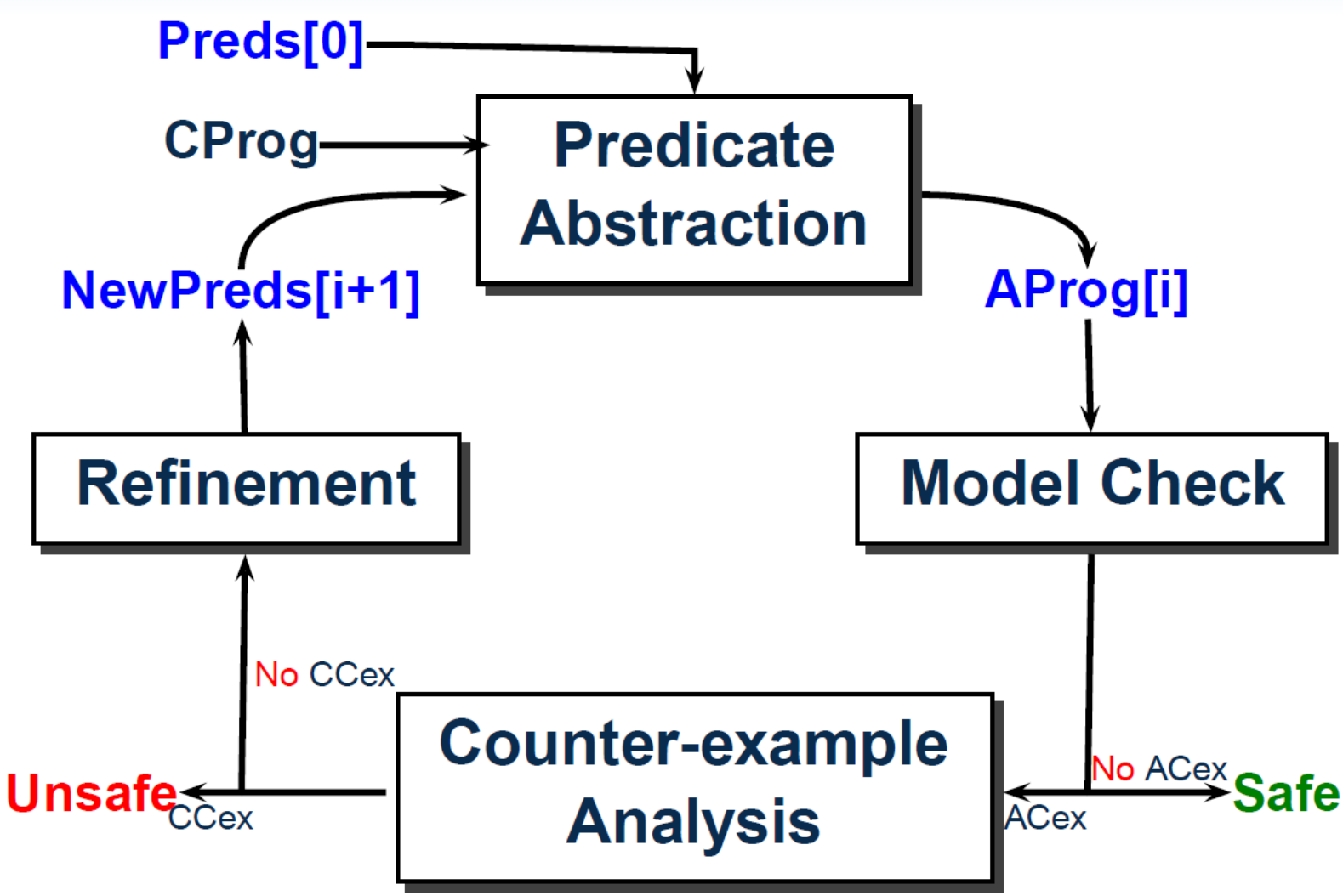
Counter-Example Guided Abstraction-Refinement (CEGAR)



Predicate abstraction



CEGAR with Predicate abstraction



Computing Abstractions

- Given concrete model CI(X), CR(X, X')
- Given set of predicates $\Psi_i(X)$
  each associated to abstract variable P_i
- Obtain the corresponding abstract model
- AR(P, P') is defined by

$$\exists X\, X'.(CR(X, X') \wedge \bigwedge_i P_i \leftrightarrow \Psi_i(X) \wedge \bigwedge_i P_i' \leftrightarrow \Psi_i(X'))$$

- Existential quantification as AllSMT
  - SMT solver extended to generate all satisfying assignment

Implicit Abstraction

- Abstract transition system computed with AllSMT:
  - Exponential in the number of predicates.
  - Major bottleneck of CEGAR.
  - Prevents the analysis of the abstract system.
- Main idea: avoid upfront computation of the abstract program
- How: embedding the abstraction definition into the BMC/k-induction encodings;
- abstract transitions implicitly computed by the SMT solver;
- similar to lazy abstraction but completely symbolic and without any image computation/quantifier elimination.

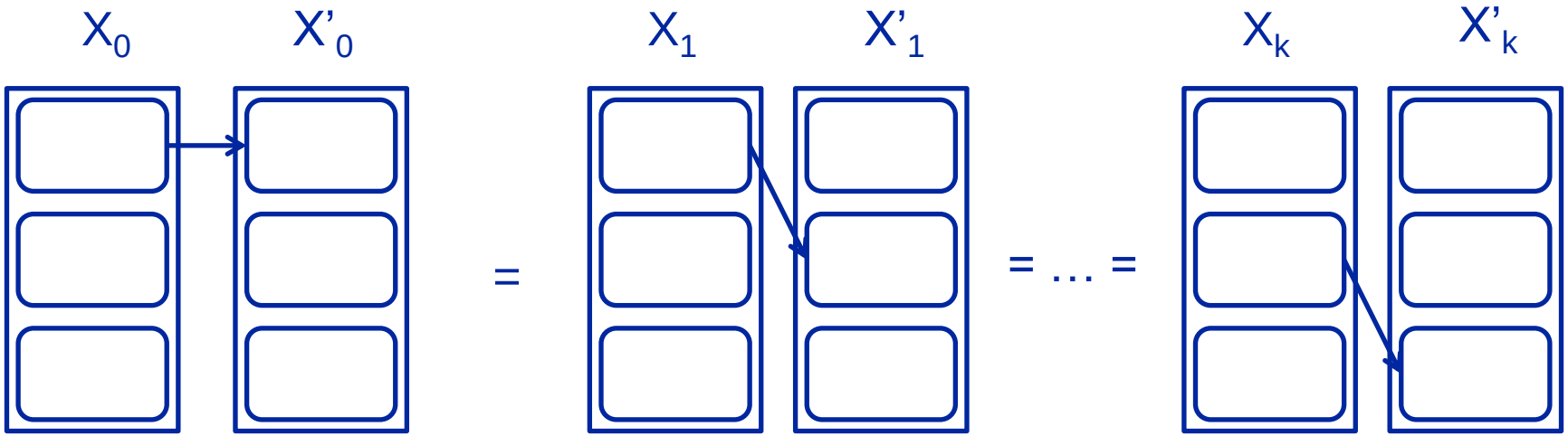
Implicit abstraction

Applicable when the abstraction α induces an equivalence relation EQ_α among the concrete states.

- For predicate abstraction,
  $EQ_\alpha(X, X') = \bigwedge_{P \in \mathcal{P}} P(X) \leftrightarrow P(X')$.

Example of application:

- Concrete unrolling: $\bigwedge_{0 \le h \le k-1} R(X_h, X_{h+1})$
- Abstract unrolling: $\bigwedge_{0 \le h \le k-1} R(X_h, X'_h) \wedge EQ_\alpha(X'_h, X_{h+1})$



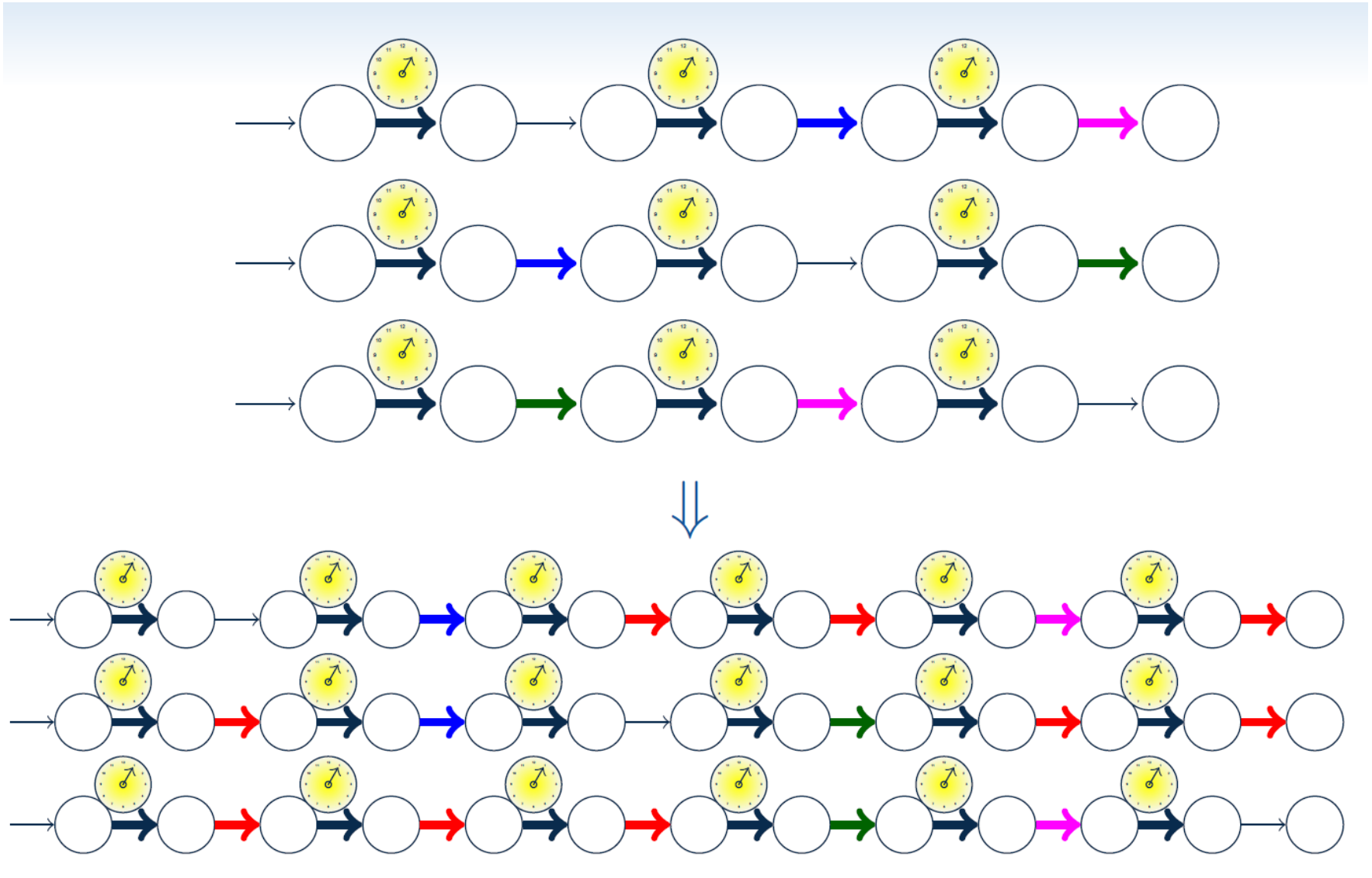
FONDAZIONE
BRUNO KESSLER

Specialized techniques

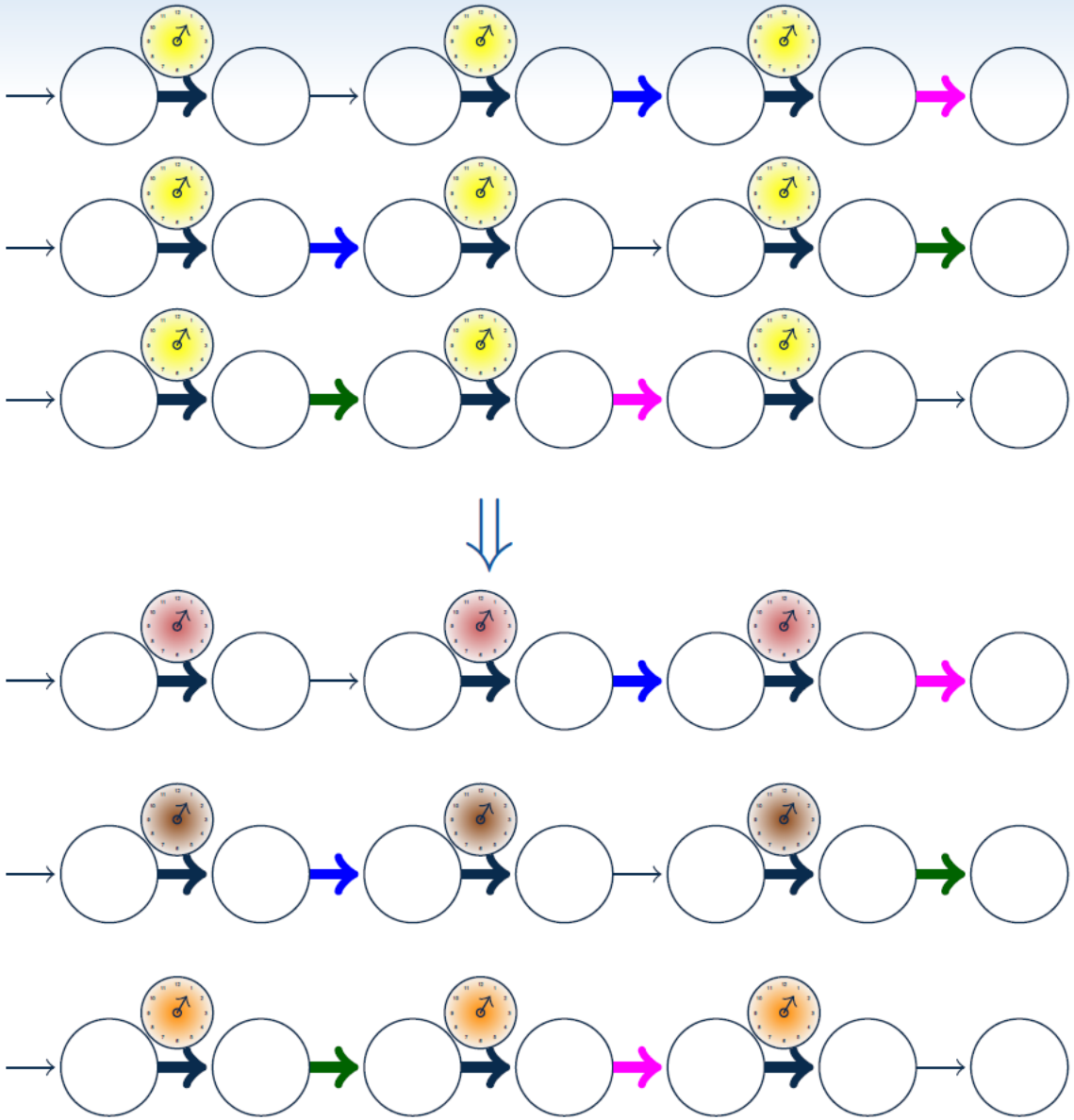
Specialized techniques

- From hybrid traces to infinite-state transition system over discrete traces
- Time elapse has the effect of a global synchronization
- Interleaving may induce very long paths
- Encoding may have significant impact!
- Generate transition systems with shorter/less paths

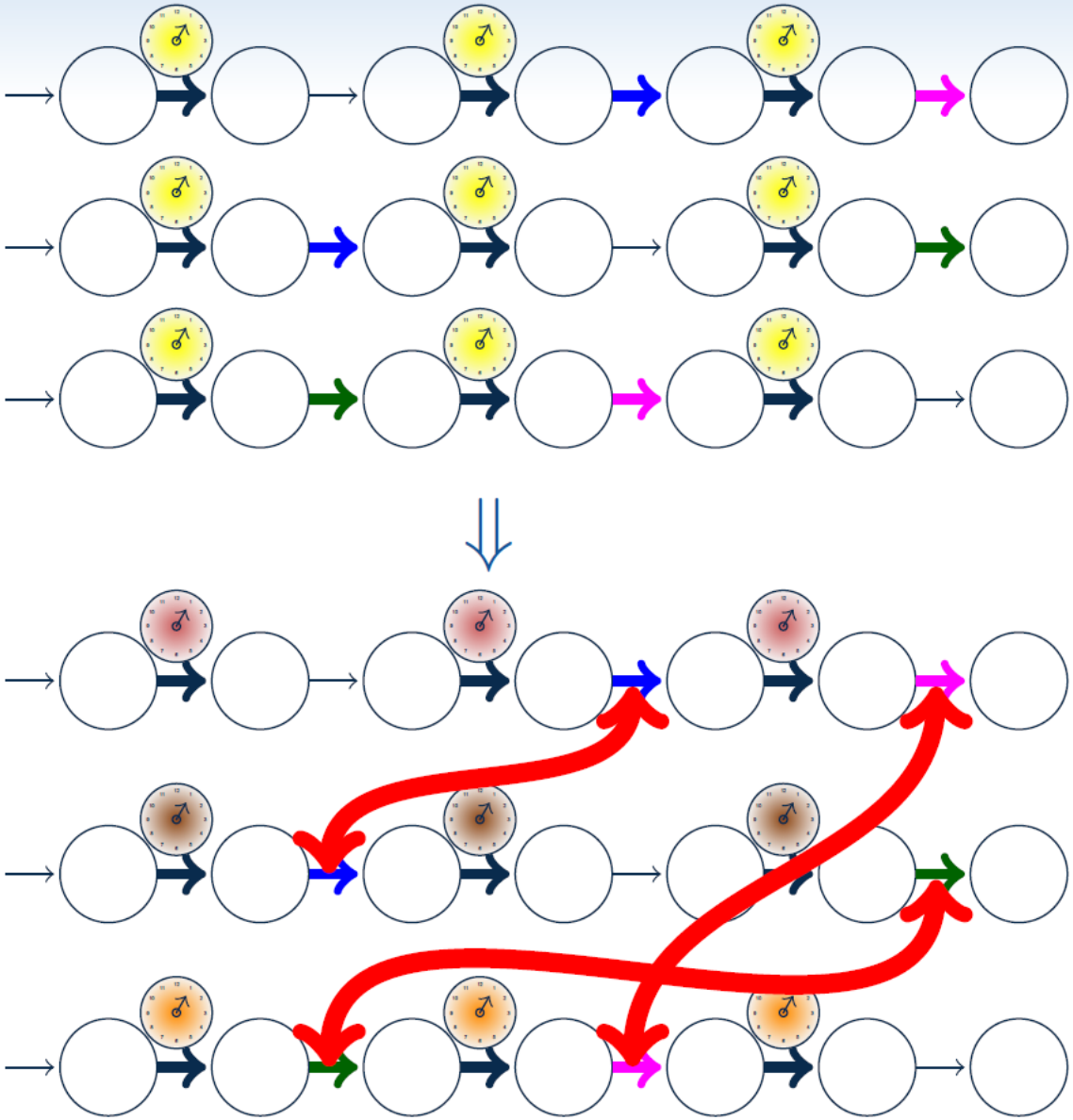
The effect of interleaving



Local clocks



Local clocks + sync constraints



Local Time Encoding

$$\textsc{Init} := \bigwedge_{q \in Q} (loc = q \rightarrow I_q(X)) \wedge t = 0$$

$$\textsc{Invar} := \bigwedge_{q \in Q} (loc = q \rightarrow Z_q(X))$$

$$\textsc{Trans} := \bigwedge_{q \in Q} (loc = q \rightarrow (\textsc{Stutter} \vee \textsc{Timed}_q \vee \bigvee_{(q,p) \in E} \textsc{Untimed}_{q,p}))$$

$$\textsc{Stutter} := \varepsilon = \mathsf{S} \wedge \delta = 0 \wedge loc' = loc \wedge X' = X \wedge t' = t$$

$$\textsc{Timed}_q := \varepsilon = \mathsf{T} \wedge \delta > 0 \wedge loc' = loc \wedge F_q(\frac{X' - X}{\delta}) \wedge t' = t + \delta$$

$$\textsc{Untimed}_{q,p} := \varepsilon = L_{q,p} \wedge \delta = 0 \wedge loc' = p \wedge J_{q,p}(X, X') \wedge t' = t$$

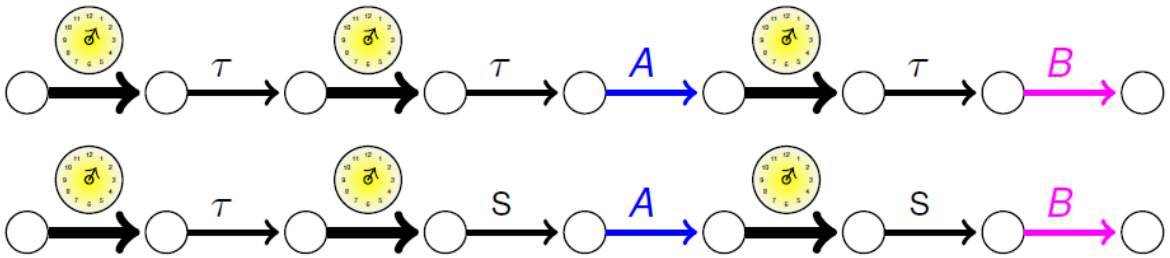
δ and T are local

Exploiting Shallow Synchronization

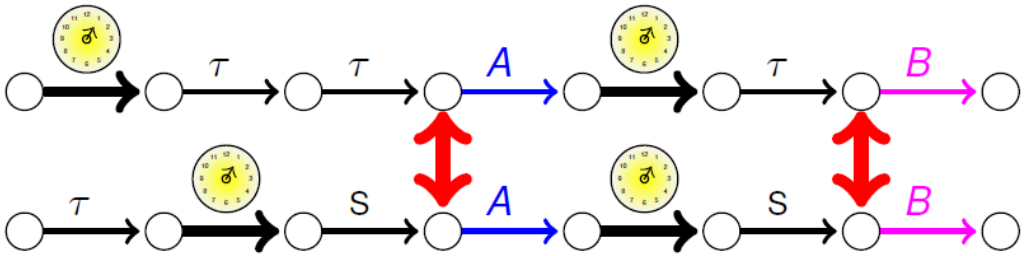
- Shallow synchronization:
  - for all systems S_j and S_h, the sequence of shared events performed by S_j and S_h is the same;
  - for all systems S_j and S_h, for all events a shared by S_j and S_h, S_j performs the i-th occurrence of a at the same time S_h performs the i-th occurrence of a;
  - for all systems S_j and S_h, the time in the last step of S_j is the same to the time in the last step of S_h.
- Different variants of the encoding:
  - Enumerating all possible combinations of occurrences.
  - Exploiting uninterpreted functions.
- Different interaction with the solver:
  - Adding sync while unrolling vs after unrolling.
  - Depth-first search vs. breadth-first search.

Possible semantics

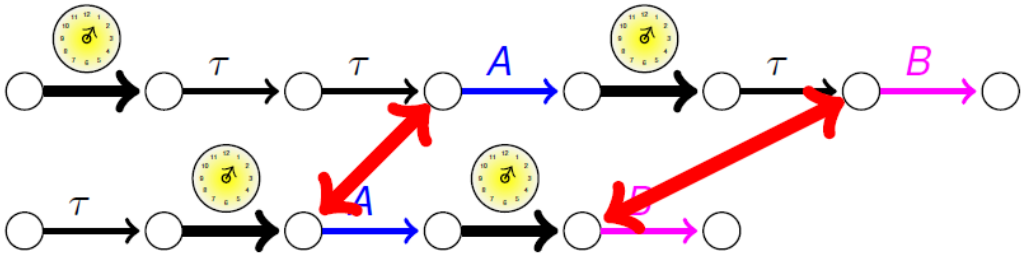
Global-time:



Local-time:



Shallow synchronization:

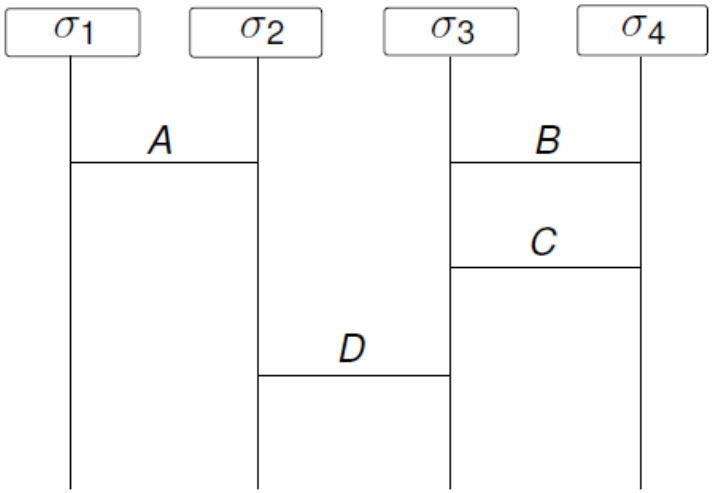


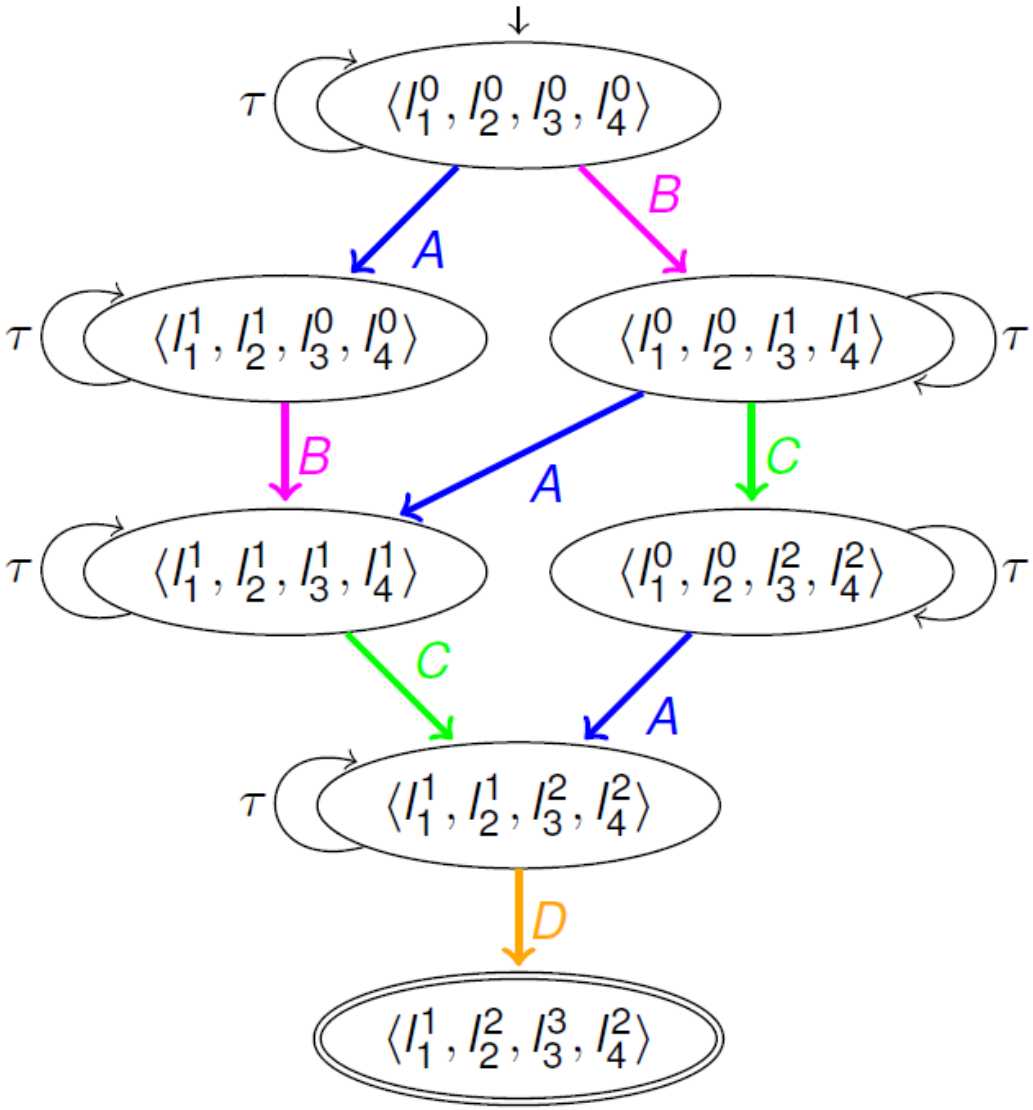
S = stutter event. τ = local event (no stutter or time).

Scenario-based verification

- A scenario is a partially specified behaviour
  - E.g. message sequence chart
- Can a scenario be refined to a concrete trace?
- A simple idea
  - encode scenario as temporal property
  - run “starndard” temporal logic model checker
- A much better idea
  - use the structure of the MSC to localize the encoding and to drive the search
  - orders of magnitude speed ups

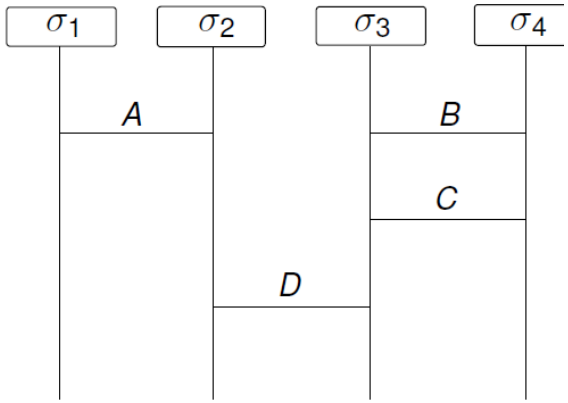
Encoding MSC into automata

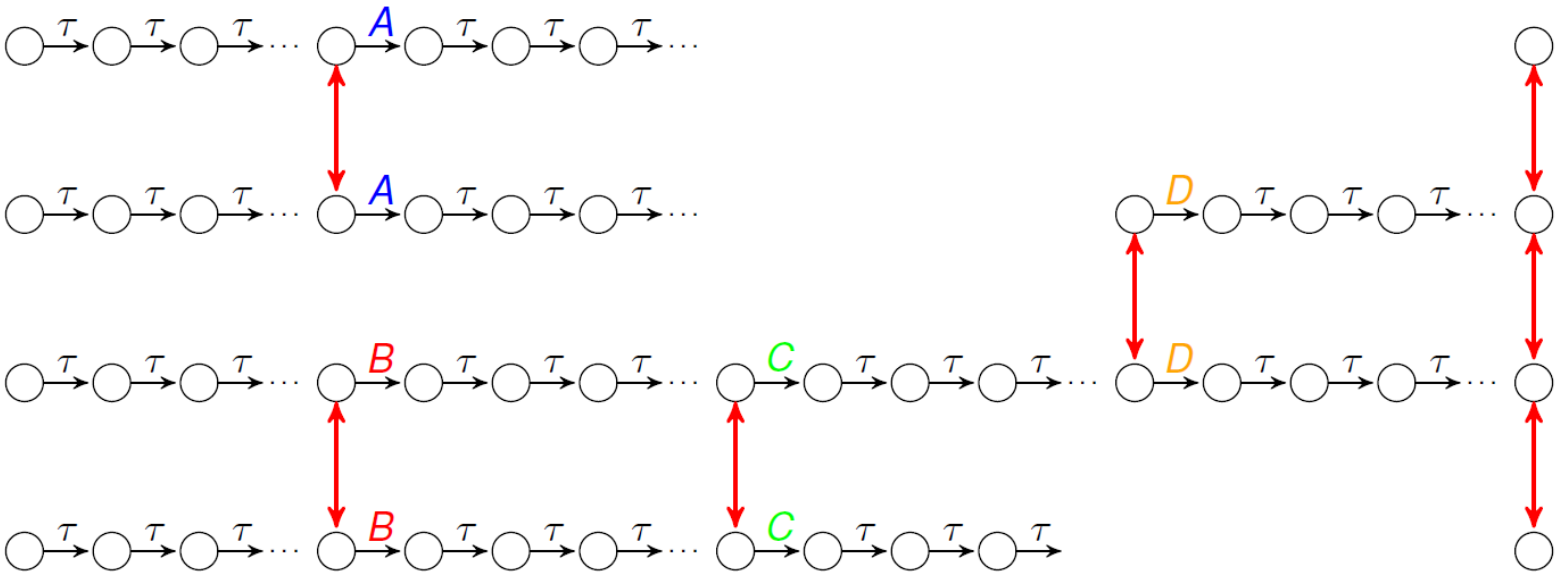




Specialized scenario encoding

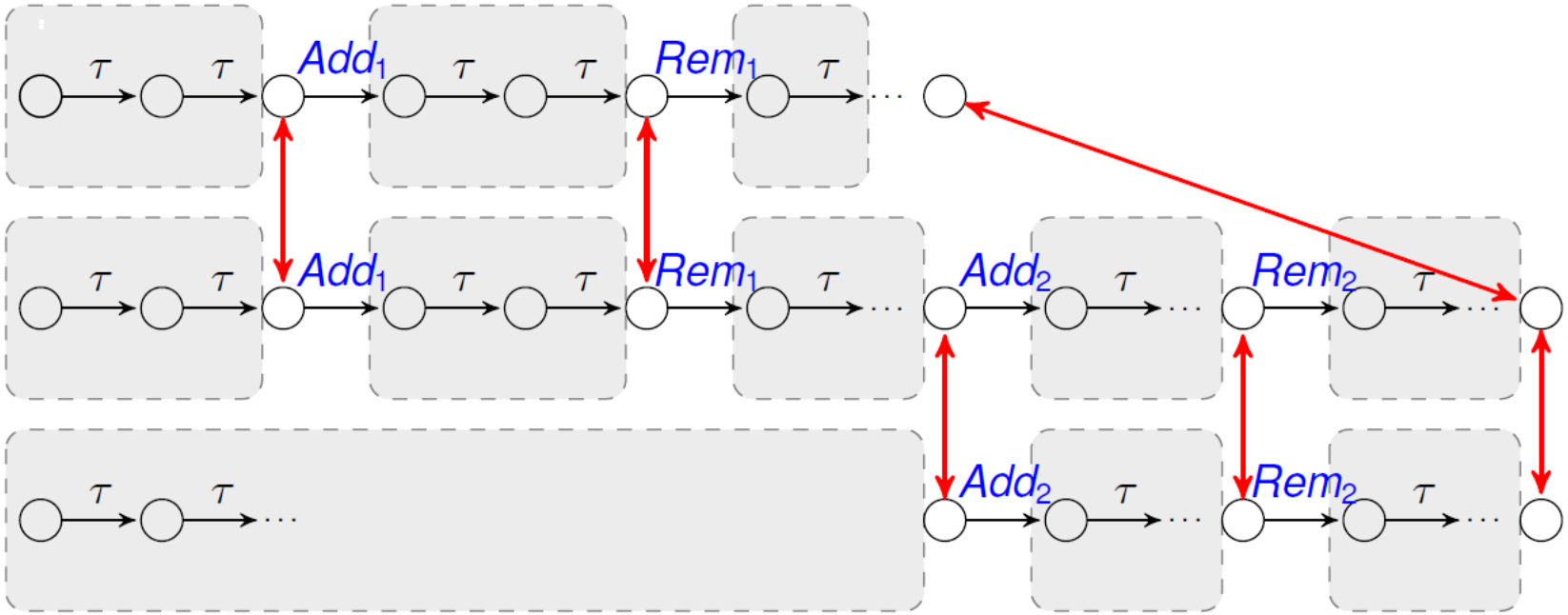
- for all the automata:
  - fix the position of the shared events.
    transition is simplified wrt shared event
  - encode the sequences of local transitions.
    transition is simplified wrt τ
  - add the synchronization constraints.





Proving unfeasibility

- Use k-induction to detect limit in expansion of sequences of local transtions



FONDAZIONE
BRUNO KESSLER

Requirements validation

Requirements are flawed

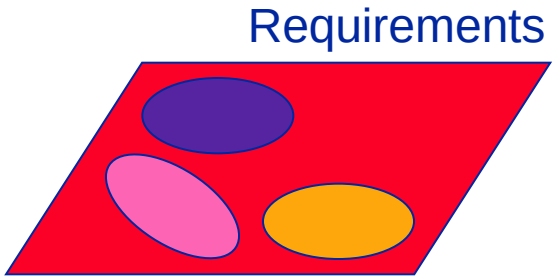
- The bugs are not in the system, but in the requirements!
  - The systems often implement correctly wrong/incomplete requirements.
  - Software system errors caused by requirements errors
- Not just a slogan, but a real user need.
- Considered as major problem of software development process by most European companies (EPRITI survey).
- Confirmed by NASA studies on Voyager and the Galileo software errors
  - Primary cause (62% on Voyager, 79% on Galileo): mis-understanding the requirements.
- Confirmed by the ESA and ERA recent calls on requirements.
- Widely acknowledged from industry across domains
  - IAI, RCF, Intecs, ...

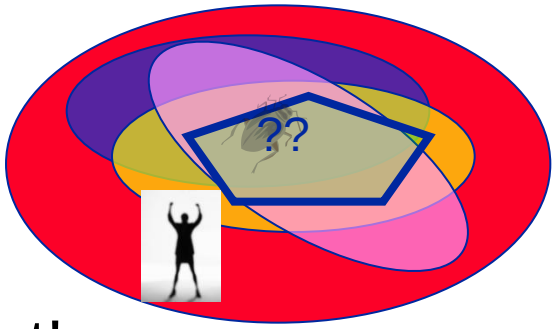
Requirements validation

- Requirements: descriptions of the functions provided by the system and its operational constraints.
- Requirements validation: checking if the requirements are correct, complete, consistent, and compliant with what the stakeholders have in mind.
- Target requirements errors:
  - Incomplete (e.g., incomplete description of a function),
  - Missing (e.g., missing assumption on lower levels),
  - Incorrect (e.g., wrong value in condition used to trigger some event),
  - Inconsistent (i.e., pair-wise incompatible),
  - Over-specified (e.g., more restrictive than necessary).
- Cover 89% of faults examined in NASA projects.

Which flaws in requirements?

- A set of requirement is a set of constraints over possible evolutions of the entities in the domain
- Possible questions
  - Are my requirements too strict?
  - Are my requirements too weak?
- Possible checks
  - Consistency check (too strict?)
    - » is there at least one admissible behaviour?
  - Possibility check (too strict?)
    - » is a given desirable behaviour admissible?
  - Assertion check (too weak?)
    - » is a given undesirable behaviour excluded?

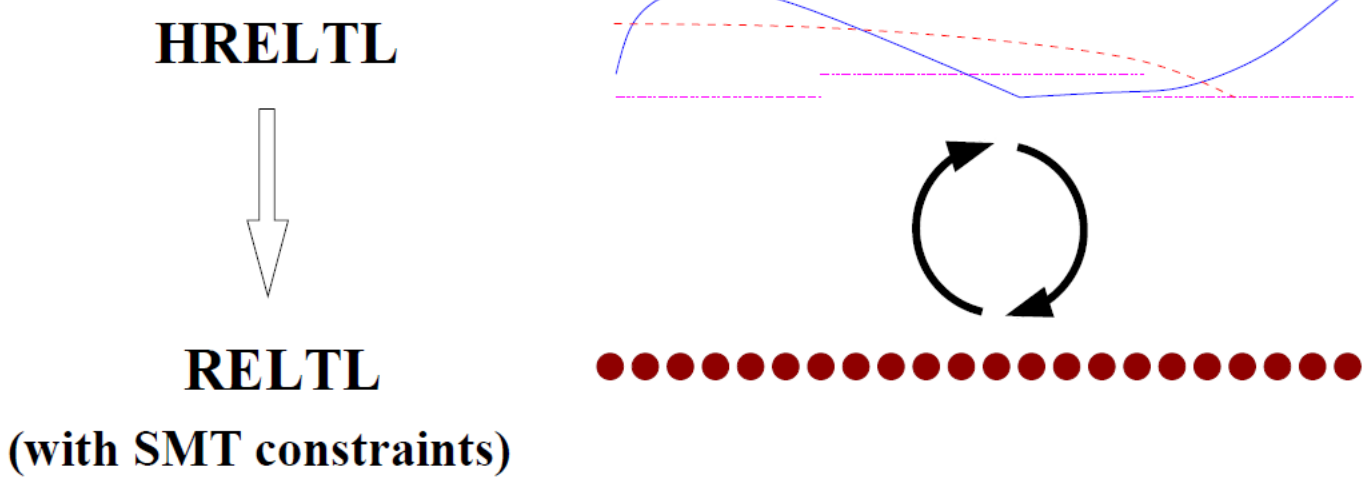




- Warning: no way to formalize design intent!

A Logic for Hybrid Traces

- HRELTL: A logic to describe hybrid traces
  - continuous and discrete evolution
  - Decision based on reduction to RELTL with SMT constraints
  - Enforce continuity by constraining values of predicates



FONDAZIONE
BRUNO KESSLER

Conclusions

Conclusions

- Hybrid Automata as an expressive and practical formalism to model complex dynamic systems
- SMT as a powerful symbolic representation formalism
  - "Model everything as one gigantic automaton? I don't think so…"
  - Well studied composition primitives
  - Structure may also help partitioning verification
- SMT solvers as powerful reasoning engines
  - to support the design phase
    - » Helping designers to gain confidence
    - » Build more predictable systems
    - » Write more reliable software
    - » Assess behaviour under faults
  - to support the operation phase
    - » Generate better plans
    - » Monitor execution
    - » Perform diagnosis
    - » Support replanning
    - » Recalibrate control strategies

Take away messages

- The need for verification
  - Very complex systems
- Verification in a broader sense
  - Rigorous analysis of the behaviour of dynamic systems
  - From off line to operation, from requirements to low level code
- Hybrid automata
  - A uniform and comprehensive formal model
- Satisfiability Modulo Theories
  - Higher level symbolic modeling
  - Efficient engines: SAT + constraint solving
- SMT-based Verification
  - Many effective complementary algorithms

Tools and applications

- The MathSAT SMT solver
  - **http://mathsat.fbk.eu**
- The NuSMV model checker
  - **http://nusmv.fbk.eu**
- A MathSAT-based extension of NuSMV
  - HyDI: a structured language for automata networks
  - **https://es.fbk.eu/tools/nusmv3/**

- Applied in
  - OMC-ARE, COMPASS, AUTOGEF, FAME, FOREVER
  - Industrial technology transfer
    - » Avionics, railways, oil and gas

Open issues and future directions

- Improving scalability of hybrid systems verification
  - Exploit structure of the problem
    - » scenario-based validation
  - Tighten connection between planning and temporal reasoning
    - » SMT-based scheduling
- Diagnosability checking and synthesis
  - Automated synthesis of sensors configurations that guarantee diagnosability
  - Generalize to the case of hybrid automata
- FDIR: fault detection, identification, recovery
  - Specification, verification and synthesis of FDIR modules
- Mixed software + physical system
  - Nasty interaction between continuous and sampled timing
    - » 100ms duty cycle with flight duration
  - Often scale very different, key is avoid trace fragmentation

FONDAZIONE
BRUNO KESSLER

Thanks for your attention

Questions?

FONDAZIONE
BRUNO KESSLER

Additional Material

FONDAZIONE
BRUNO KESSLER

Some interesting applications

Applications to High-level HW Design

- Ongoing work with Intel Haifa
  - Application described in "high level" language
  - words and memories are not blasted into bits
- Custom decision procedure for Bit Vectors
- Applications
  - Register-transfer level circuits
  - Microcode
- Functionalities
  - more scalable verification
    - » currently based on boolean SAT
  - tight integration with symbolic simulation
    - » pipe of proof obligations
  - Automated Test Pattern Generation
    - » enumerate many different randomized solutions
- Results
  - MathSAT currently “in production”
    - » Integrated in design environment deployed to microcode engineers
  - Best paper award at FMCAD’10

Analysis of Railways Control Software

- Control software for Interlocking
  - controls devices in train station
  - Application independent scheduler
  - Parameterized, object oriented
  - Instantiation with respect to station topology

- Model Checking to analyze single modules
  - SMT-based software model checking
  - checking termination, functional properties
- Compositional reasoning for global proofs
  - based on scheduler structure
- Reverse engineering from the code
  - inspection, what-if reasoning

- Other potential role of SMT solving
  - dealing with quantified formulae over lists of entities

Parametric Schedulability Analysis

- Schedulability analysis
  - given set of processes and scheduling policy
  - check whether deadlines can be met
- Key problem: sensitivity analysis
  - where do the numbers come from?
  - typically, these are estimates
  - traditional schedulability theory based on numerical raesoning, lifting results to practical cases may be nontrivial
- Goal: analyze sensitivity with respect to variations
- Analytical construction of schedulability region!
- The role of SMT
  - SMT allows for parametric representation
  - SMT-based bounded model checking to generate one fragment of unschedulability region
  - iterate to generate all fragments
  - CEGAR to terminate the iteration

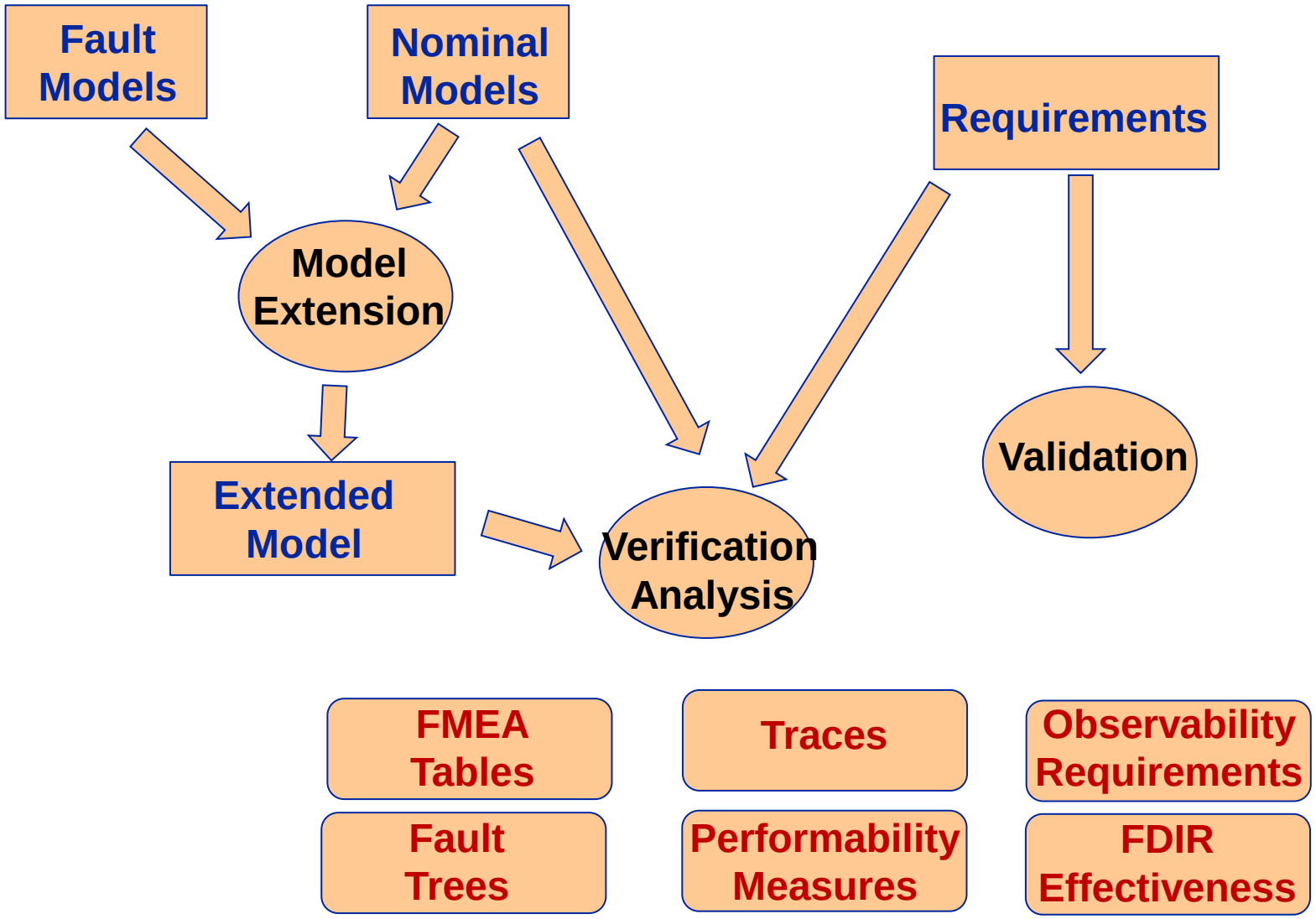
Design Mutation

- The problem: find "good" spatial position of aircraft components with respect to safety constaints
  - no electrical components "below" component that potential leakage
  - not all components implementing critical function on same impact trajectory
- Required functionalities
  - is a configuration satisfactory
  - reasons for violation
  - find acceptable solution
  - find optimal solution
- Encode problem into SMT
  - may require dedicated, custom theory
  - may require extension to "optimal constraints"

FONDAZIONE
BRUNO KESSLER

A design flow based on Formal Methods

The flow of design phase



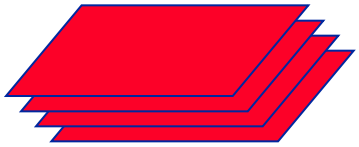
Requirements Validation

- The error is in the requirements, not in the system
  - a real user need
- Validate system requirements *before* detailed design and implementation
  - *"Are we capturing the right system?"*
- Available functionalities:
  - Property simulation
  - Check logical consistency
    - » Are there any contradictions?
  - Check property strictness
    - » Are the properties strict enough to rule out undesired behaviours?
  - Check property weakness
    - » Are the properties weak enough to allow desirable behaviours?

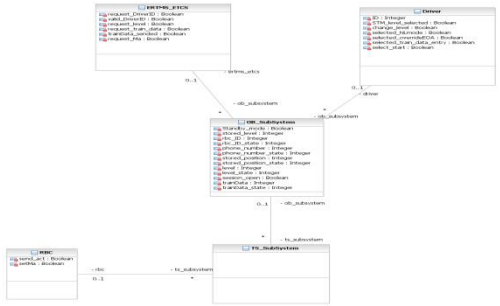
- A whole research line on its own:
  - Temporal logic satisfiability engines
  - Diagnostic information: unsatisfiable cores
  - Relevant projects
    - » Formal requirements validation of European Train Control System [ERA]
    - » OthelloPlay [MRS research award]

Requirements: Informal to Formal

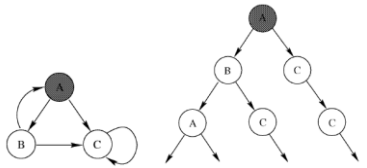
NATURAL LANGUAGE

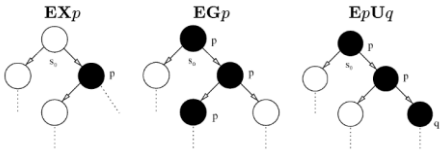


SEMIFORMAL LANGUAGE



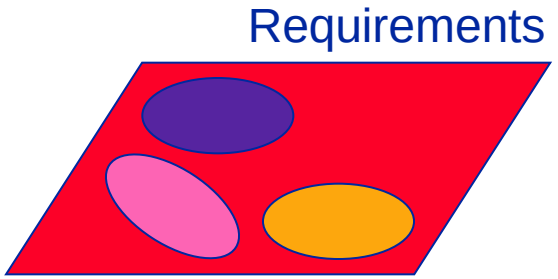
FORMAL LANGUAGE

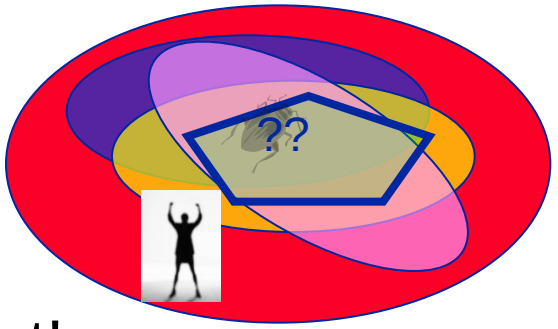




Which flaws in requirements?

- A set of requirement is a set of constraints over possible evolutions of the entities in the domain
- Possible questions
  - Are my requirements too strict?
  - Are my requirements too weak?
- Possible checks
  - Consistency check (too strict?)
    - » is there at least one admissible behaviour?
  - Possibility check (too strict?)
    - » is a given desirable behaviour admissible?
  - Assertion check (too weak?)
    - » is a given undesirable behaviour excluded?





- Warning: no way to formalize design intent!

Functional Correctness

- Correctness verification
  - *“Are we building the system right?”*
- Available functionalities:
  - Model Simulation
    - » Animate model to produce execution traces
  - Property Verification
    - » Check that a specification holds in all model traces
    - » E.g. *“always (voltage >= 5.8)”*

Safety Analysis

- Safety analysis
  - Evaluate hazards and risks
  - Check system behavior in presence of faults

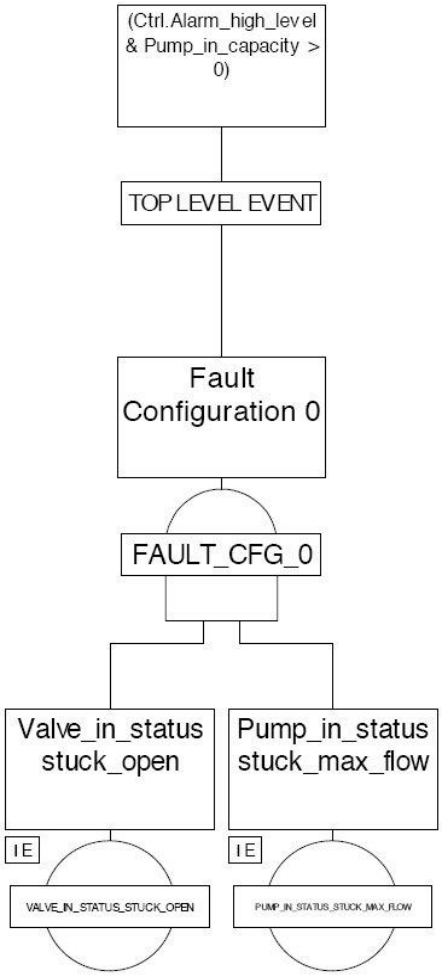
- Modeling combined nominal and faulty behaviour:
  - Nominal model annotated with possible faults
    - » “Valve stuck at open”, “jammed engine”
  - Select model behaviour under fault
    - » E.g. “constant value”, “ramp down until stop”
  - Combined behaviour automatically extended
    - » Fault variables model presence of faults
    - » Mutiplex nominal/faulty behaviour

- Analyses:
  - Fault Tree Analysis (FTA)
  - Failure Modes and Effects Analysis (FMEA)

- Based on the FSAP tool
  - Various UE projects: ESACS, ISAAC, MISSA
  - Recent book on topic [BV10]:

Safety Analysis

- Fault Tree Analysis (FTA)
  - Find the minimal combinations of faults that may cause a top event
    - » E.g.: "*Which combinations of faults may cause alarm to be raised*"



- Reduction to parametric model checking
  - Parameters are failure mode variables
  - Intuition:
    - » Find violation to property
    - » Extract assignment to fault variables
    - » Accumulate, block, and iterate until fix point

Safety Analysis

- Failure Modes and Effects Analysis (FMEA)
  - Analyze the impact of fault configurations on a set of system properties
    - » E.g. *"What are the consequences of a battery failure: i) on the output voltage of the power generator? ii) on the output alarm?"*

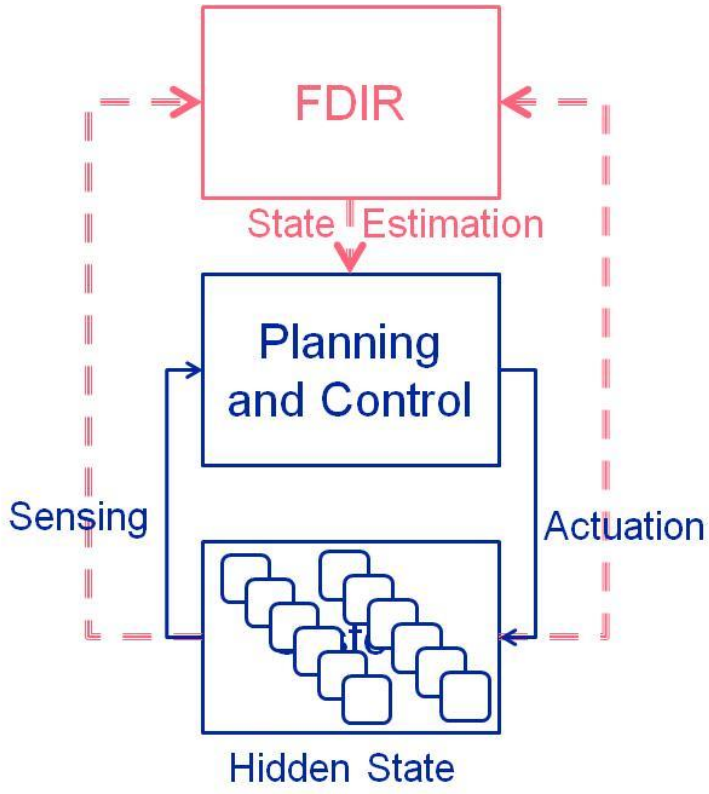
Ref. No.	*Item*	*Failure mode*	*Failure cause*	*Local effects*	*System effects*	*Detection means*	*Severity*	*Corrective Actions*
1	Pump	Fails to operate	Comp. broken No input flow	Coolant temperature increases	Reactor temperature increases	Temperature alarm	Major	Start secondary pump Switch to secondary circuit
2	Valve	Stuck closed	Comp. broken	Excess liquid	Reactor pressure increases	Coolant level sensor	Critical	Open release valve
3		Stuck open	Comp. broken	Insufficient liquid	Reactor temperature increases	Coolant level sensor	Critical	Open tank valve

- Reduction to model checking
  - Failure mode variables suitably constrained
  - Simplify extended model
  - Solve multiple properties in simplified model

FDIR effectiveness analysis

- Fault Detection
  - *"Will given FDIR procedure always detect a fault?"*
- Fault Isolation
  - *"Will given FDIR procedure identify the fault responsible for an event?"*
- Fault Recovery
  - *"Will given FDIR procedure recover from a fault?"*

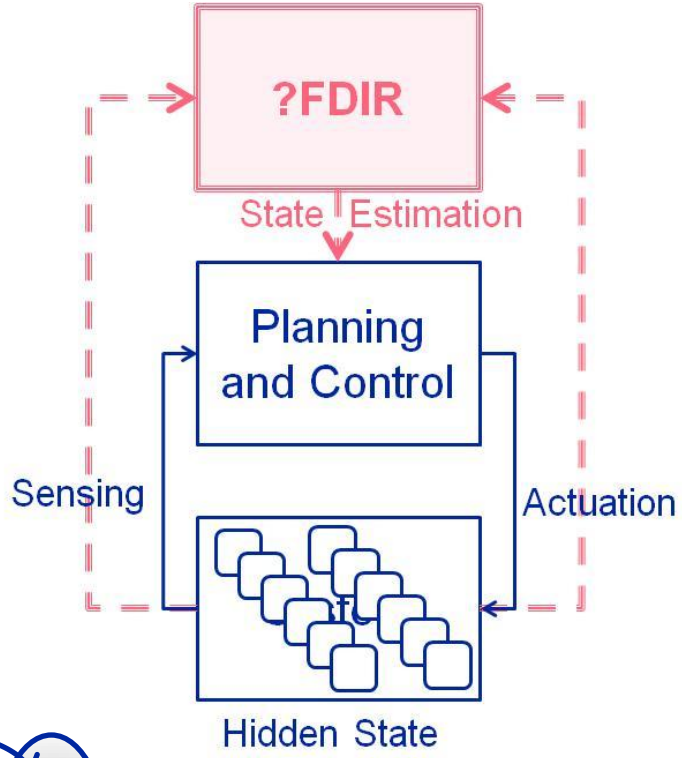
- Solved by direct reduction to model checking of extended model
  - Analysis of closed loop behaviour
    - » system + controller + FDIR

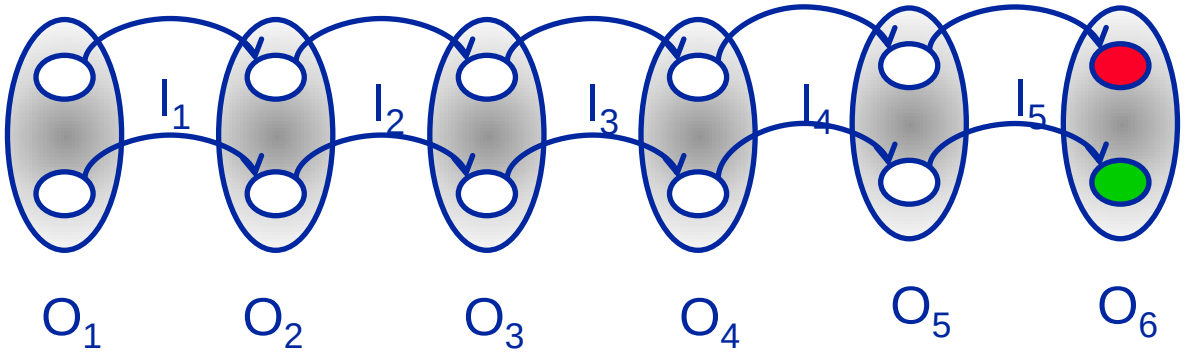


Diagnosability Analysis

- Diagnosis feasibility
  - *"Is there a diagnoser for a given property?"*
- Diagnoser synthesis
  - *"Find a good sensors configuration"*

- Diagnosability re-cast to model checking in the twin plant model:
  - *Twin plant: synchronous product of the model of the plant with itself imposing equality of the actions and of the observations*
  - *There is no pair of execution one reaching a bad state, the other reaching a good state, with identical observations*





Designing FDIR

- A very important problem
- Currently no adequate methodologies for FDIR
- AUTOGEF
  - Formal requirements specification for FDIR components
    - » Correctness – raise alarms only when required
    - » Completeness – raise alarms whenever required
      - What if not diagnosable?
  - Verification and synthesis of FDIR modules
- FAME
  - Take into account timed fault propagation
- HASDEL
  - Application to launchers

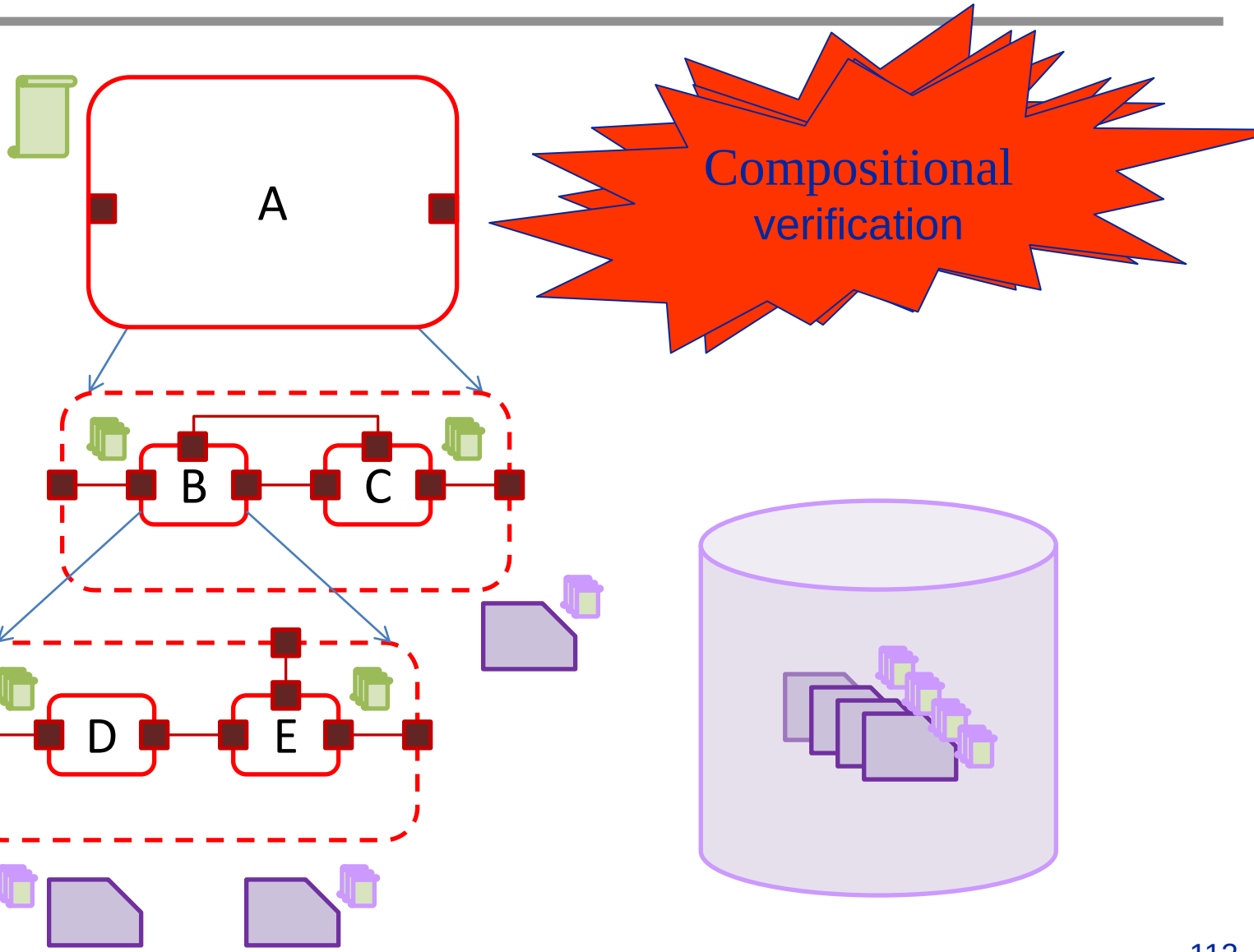
FONDAZIONE
BRUNO KESSLER

Contract-based Design

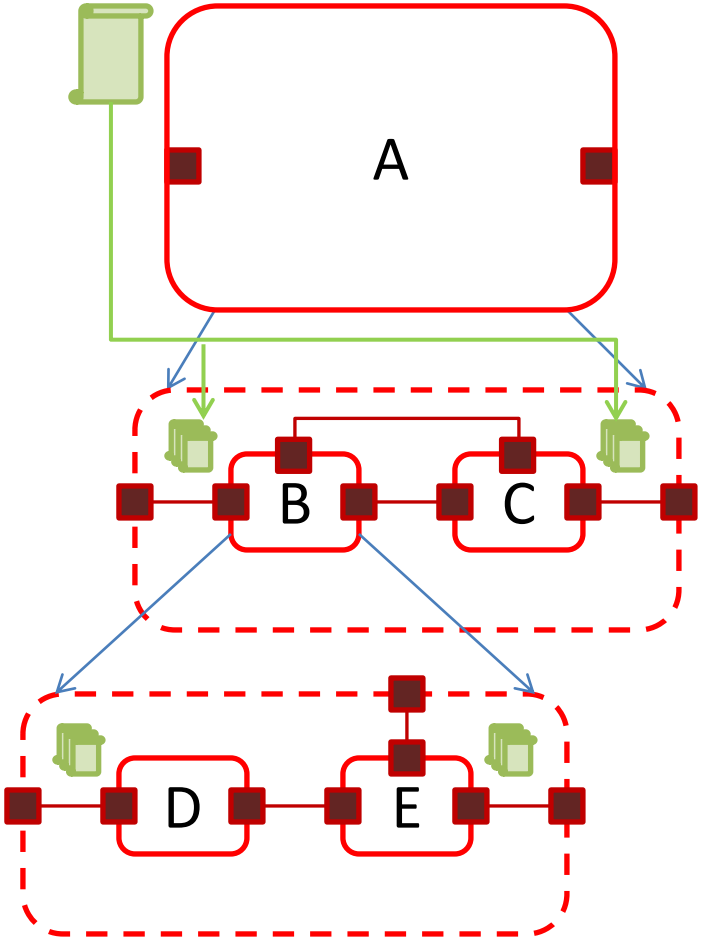
Contract-based design

- Modeling of a space systems supporting:
  - Functional step-wise refinement
  - From system to software
  - Exploiting the SRA
- FoReVer adopts a component-based approach to:
  - Describe the architectural blocks of the system.
  - Consider such blocks as black boxes until they are refined.
  - Identify the SRA parts that can be reused.
- FoReVer adopts a contract-based design to:
  - Formalize properties of system and components distinguishing between assumption and guarantees.
  - Formalize the guarantees provided by the SRA and the correct reuse of SRA components.
  - In general, to support:
    - » Step-wise refinement
    - » Compositional verification
    - » Reuse of components

Contract-based approach



Refinement



- Component decomposed into subcomponents
- Contract refined into collection of contracts over subcomponents
- Contract refinement can be formally proved
  - Contracts as formulae
  - Correctness of refinement as validity checking of proof obligations
- Formal check within OCRA

Correctness

- The FoReVer model is correct iff
  - For every refined contract, the refinement is correct.
  - For every state machine, the state machine is a correct implementation of the component's contracts.

EagleEye example

- First collected info on the system physical architecture.
- Identified FDIR requirements to detail system-to-software refinement.
- Decomposed in one requirement for each type of anomaly:
  - Critical Values Reading
  - Alive Flag Failure
  - Consistency Check Failure
  - TC/TM Correctness
  - TC failed execution
- Chosen Critical Value as first example to exercise the methodology and the tool support.

FDIR Critical Values

- Monitoring a critical variable.
- Triggering an alarm when the value reaches a threshold.
- More complex checks can be formalized:
  - Ranges or delta variation or expected value.
  - Alarm can be triggered after repeated checks.
- When the alarm is triggered, move to SHM to be controlled by ground.
- More complex recovery can be formalized:
  - First try reconfiguration procedure.
- 4 architectures formalized in FoReVer and enriched with a contract refinement.
- In the software architecture, the SRA pseudo-components have been defined with their contracts.
- These components and contracts will be reused in the GB2 case study.