Systems Design Laboratory

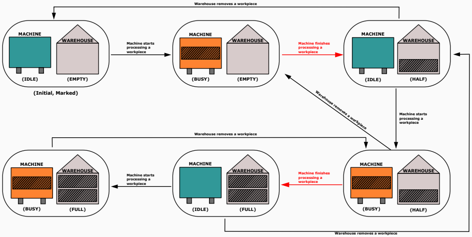
Extended Finite (State) Automata

[1]Department of Mathematics, University of Padova, ITALY

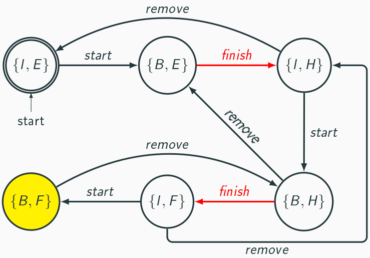
[2]Department of Computer Science, University of Verona, ITALY

Plant Level Synchronization

Graphical Description


Warehouse removes a workpiece
MACHINE
WAREHOUSE
(IDLE)
(EMPTY)
(Initial, Marked)
Machine starts processing a workpiece
MACHINE
WAREHOUSE
(BUSY)
(EMPTY)
Machine finishes processing a workpiece
MACHINE
WAREHOUSE
(IDLE)
(HALF)
Warehouse removes a workpiece
Machine starts processing a workpiece
Warehouse removes a workpiece
MACHINE
WAREHOUSE
(BUSY)
(FULL)
Machine starts processing a workpiece
MACHINE
WAREHOUSE
(IDLE)
(FULL)
Machine finishes processing a workpiece
MACHINE
WAREHOUSE
(BUSY)
(HALF)
Warehouse removes a workpiece

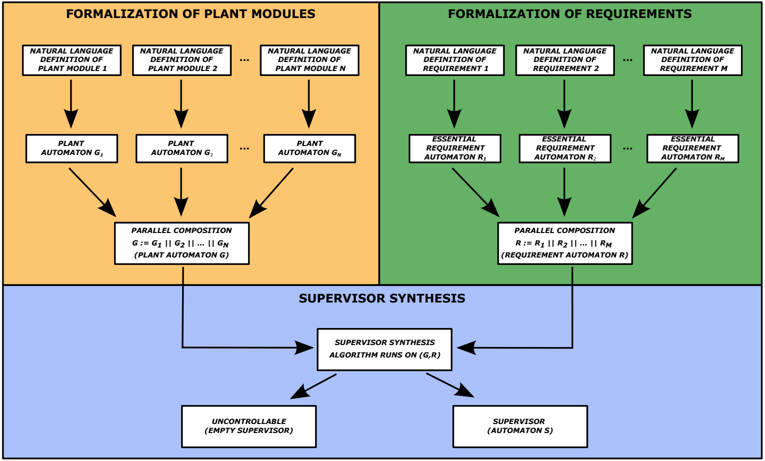
Parallel composition



- When the machine is BUSY and the warehouse is FULL, despite that *finish* is uncontrollable, the machine cannot execute it since *finish* is not executable by the warehouse.

Plant level synchronization can prevent (uncontrollable) events from being executed.

Supervisor Synthesis: Workflow



FORMALIZATION OF PLANT MODULES

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 1

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 2

...

NATURAL LANGUAGE DEFINITION OF PLANT MODULE N

PLANT AUTOMATON G_1

PLANT AUTOMATON G_2

...

PLANT AUTOMATON G_N

PARALLEL COMPOSITION $G := G_1 \,||\, G_2 \,||\, \dots \,||\, G_N$ (PLANT AUTOMATON G)

FORMALIZATION OF REQUIREMENTS

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 1

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 2

...

NATURAL LANGUAGE DEFINITION OF REQUIREMENT M

ESSENTIAL REQUIREMENT AUTOMATON R_1

ESSENTIAL REQUIREMENT AUTOMATON R_2

...

ESSENTIAL REQUIREMENT AUTOMATON R_M

PARALLEL COMPOSITION $R := R_1 \,||\, R_2 \,||\, \dots \,||\, R_M$ (REQUIREMENT AUTOMATON R)

SUPERVISOR SYNTHESIS

SUPERVISOR SYNTHESIS ALGORITHM RUNS ON (G,R)

UNCONTROLLABLE (EMPTY SUPERVISOR)

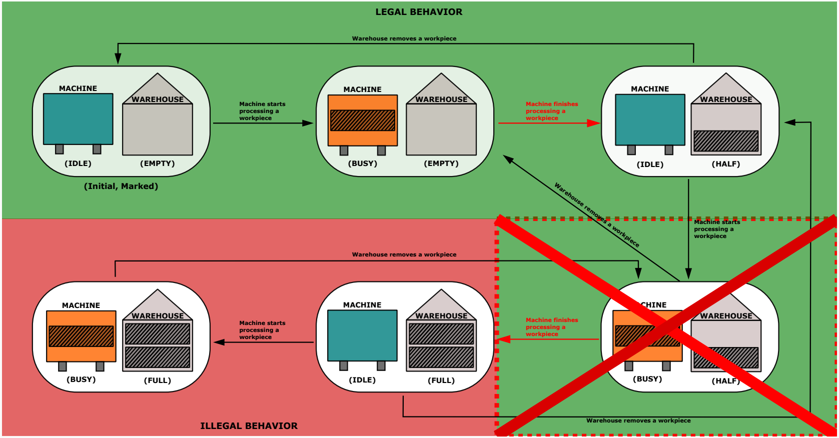
SUPERVISOR (AUTOMATON S)

Warning: in the parallel $G||H$ we need:

1. Remove non-coaccessible states (blocking)
2. Remove states such that uncontrollable events are allowed by the plant but not by the requirement.

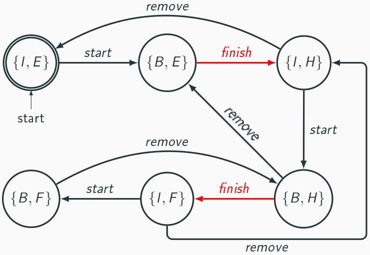
Machine-Warehouse example

Requirement 1: The warehouse stores at most one workpiece

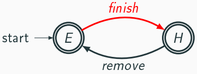

LEGAL BEHAVIOR
Warehouse removes a workpiece
MACHINE
WAREHOUSE
(IDLE)
(EMPTY)
(Initial, Marked)
Machine starts processing a workpiece
MACHINE
WAREHOUSE
(BUSY)
(EMPTY)
Machine finishes processing a workpiece
MACHINE
WAREHOUSE
(IDLE)
(HALF)
Warehouse removes a workpiece
Machine starts processing a workpiece
Warehouse removes a workpiece
MACHINE
WAREHOUSE
(BUSY)
(FULL)
Machine starts processing a workpiece
MACHINE
WAREHOUSE
(IDLE)
(FULL)
Machine finishes processing a workpiece
CHINE
WAREHOUSE
(BUSY)
(HALF)
Warehouse removes a workpiece
ILLEGAL BEHAVIOR

Example 1: Synthesis Algorithm - Tentative Supervisor

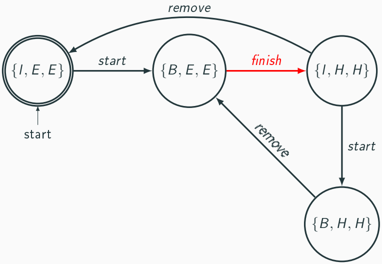
Plant Automaton G



Requirement R_1

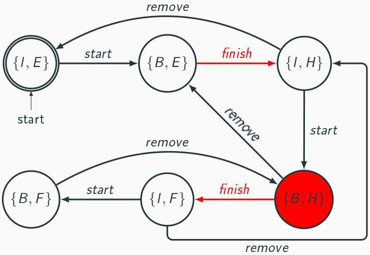


Parallel composition $G||R_1$

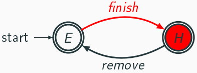


Example 1: Synthesis Algorithm - Removal of states

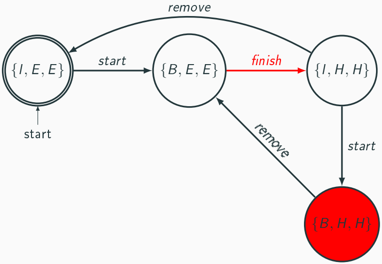
Plant Automaton G



Requirement R_1



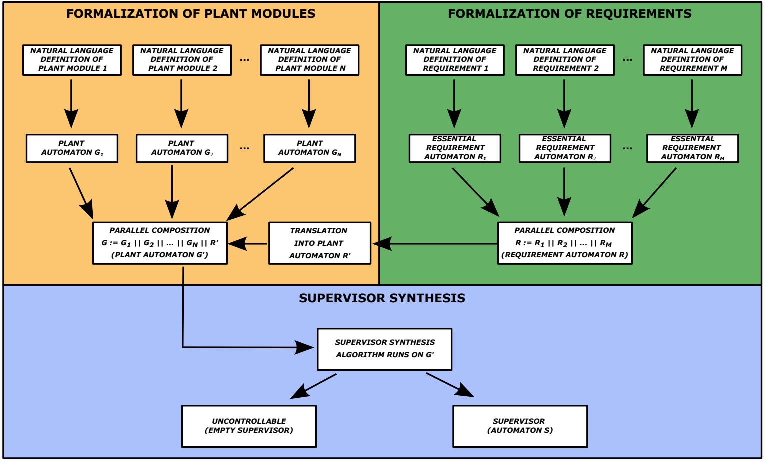
Parallel composition $G \| R_1$



We remove $\{B, H, H\}$ since $\{B, H\}$ (plant) enables *finish*, whereas $\{H\}$ (requirement) does not.

Can we modify the approach to model the executions of G that are forbidden by R as blocking problems?

Supervisor Synthesis: Forbidden Executions as Blocking


FORMALIZATION OF PLANT MODULES

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 1

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 2

...

NATURAL LANGUAGE DEFINITION OF PLANT MODULE N

PLANT AUTOMATON G_1

PLANT AUTOMATON G_2

...

PLANT AUTOMATON G_N

PARALLEL COMPOSITION $G := G_1 \mid\mid G_2 \mid\mid \ldots \mid\mid G_N \mid\mid R'$ (PLANT AUTOMATON G')

TRANSLATION INTO PLANT AUTOMATON R'

FORMALIZATION OF REQUIREMENTS

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 1

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 2

...

NATURAL LANGUAGE DEFINITION OF REQUIREMENT M

ESSENTIAL REQUIREMENT AUTOMATON R_1

ESSENTIAL REQUIREMENT AUTOMATON R_2

...

ESSENTIAL REQUIREMENT AUTOMATON R_M

PARALLEL COMPOSITION $R := R_1 \mid\mid R_2 \mid\mid \ldots \mid\mid R_M$ (REQUIREMENT AUTOMATON R)

SUPERVISOR SYNTHESIS

SUPERVISOR SYNTHESIS ALGORITHM RUNS ON G'

UNCONTROLLABLE (EMPTY SUPERVISOR)

SUPERVISOR (AUTOMATON S)

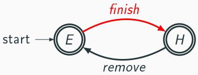
Supervisor synthesis can be seen as a modified version of trim that also takes into consideration the removal of uncontrollable events from G'. This way, we only need to reason on G'.

Forbidden Plant Executions as Blocking Problems

Problem:

- **Input:** A plant G and a requirement R.
- **Output:** A requirement R' that models the "forbidden" executions of G in a way that such executions will appear as blocking problems in $G \| R'$.

Requirement R_1

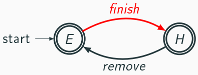


$\rightsquigarrow$???

Forbidden Plant Executions

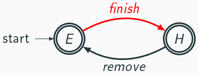
Start with a copy of R_1.

Requirement R_1



$\rightsquigarrow$

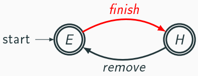
Requirement R_1'



Forbidden Plant Executions as Blocking Problems

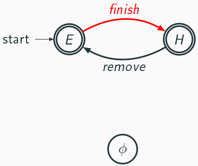
Add a forbidden state ϕ. Leave the state unmarked.

Requirement R_1



$\rightsquigarrow$

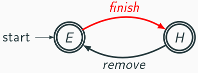
Requirement R_1'



Forbidden Plant Executions as Blocking Problems

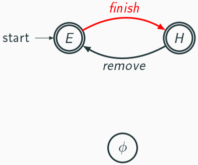
For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.

Requirement R_1



$\rightsquigarrow$

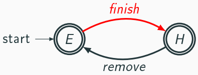
Requirement R_1'



Forbidden Plant Executions as Blocking Problems

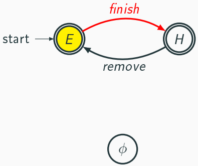
For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.

Requirement R_1



$\rightsquigarrow$

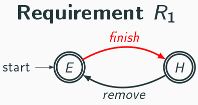
Requirement R_1'



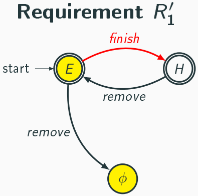
What about state E?

Forbidden Plant Executions as Blocking Problems

For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.



$\rightsquigarrow$

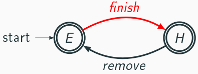


Add a transition $E \rightarrow \phi$ labeled by *remove*.

Forbidden Plant Executions as Blocking Problems

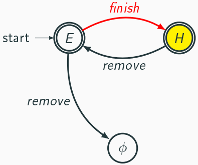
For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.

Requirement R_1



$\rightsquigarrow$

Requirement R_1'

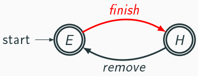


What about state H?

Forbidden Plant Executions as Blocking Problems

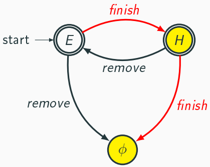
For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.

Requirement R_1



$\rightsquigarrow$

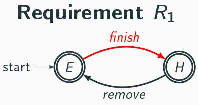
Requirement R_1'



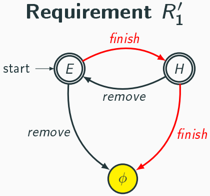
Add a transition $H \rightarrow \phi$ labeled by *finish*.

Forbidden Plant Executions as Blocking Problems

For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.



$\rightsquigarrow$

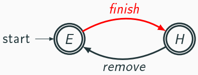


What about ϕ itself?

Forbidden Plant Executions as Blocking Problems

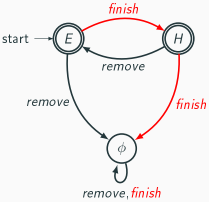
For each state s of R_1' and each event e of R_1', if e cannot be executed from s add a transition from s to ϕ labeled by e.

Requirement R_1



$\rightsquigarrow$

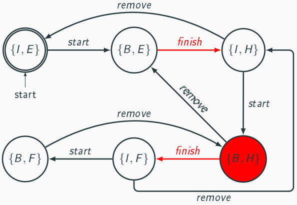
Requirement R_1'



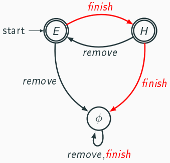
Add self-loops transitions for all events of R_1' (special cases of the statement above: "for each state" = ϕ included).

The parallel composition $G \| R_1'$

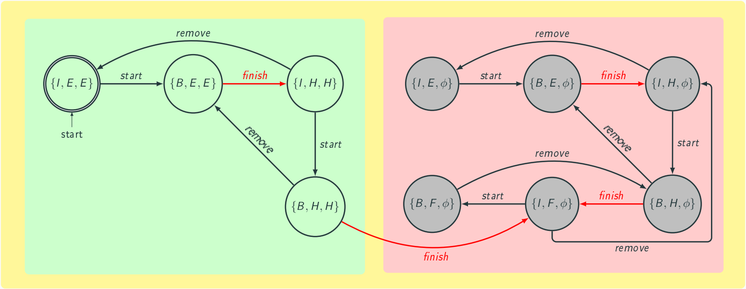
Plant G



Requirement R_1'



Executions of G allowed by R_1' (i.e., $G \| R_1'$)

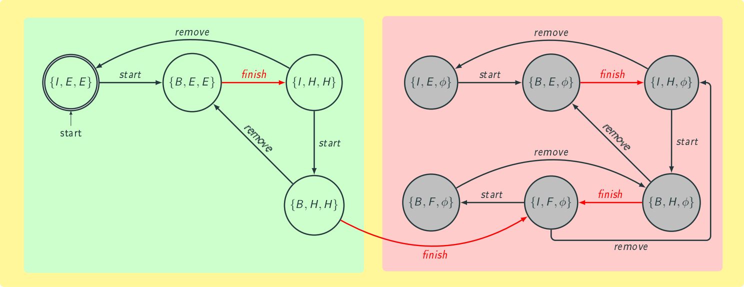


Executions of G allowed by R_1

Executions of G forbidden by R_1

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



Executions of G allowed by R_1

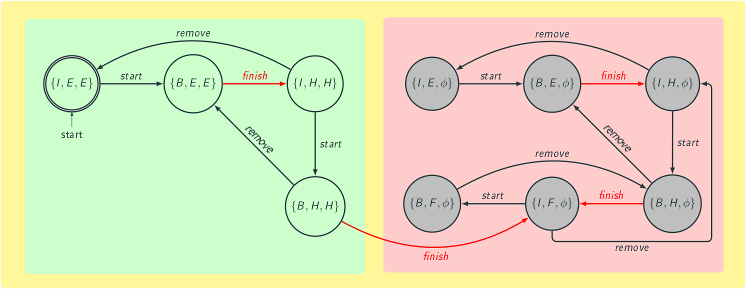
Executions of G forbidden by R_1

What next? Can you think about a straightforward algorithm to synthesize a supervisor (or prove than none exists)?

Forbidden states

Some states are more equal than others.

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



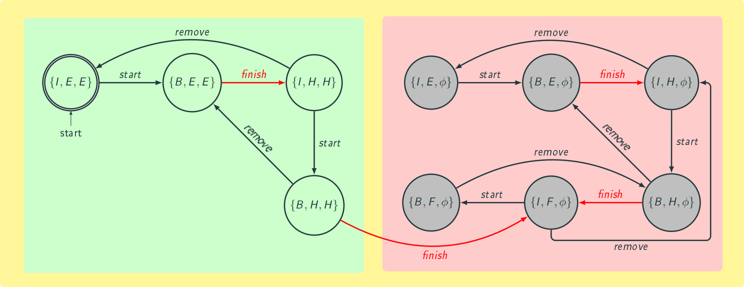
Executions of G allowed by R_1

Executions of G forbidden by R_1

$$Forbidden(G \| R_1') := \{(g, r) \in States(G \| R_1') \mid r = \phi\}$$

Borderline forbidden states

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



Executions of G allowed by R_1

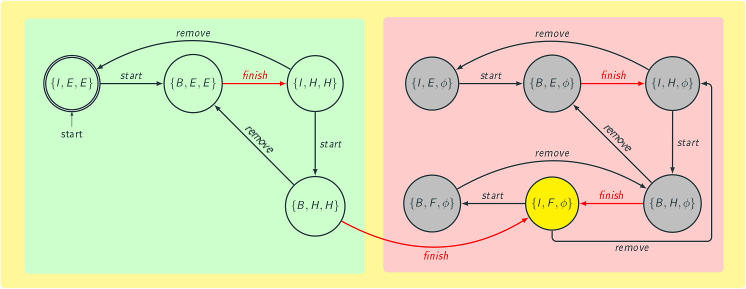
Executions of G forbidden by R_1

A forbidden state (g, ϕ) is called **border forbidden** if there exists a non-forbidden state (g', r') from which we can reach (g, ϕ) by executing some transition.

What is/are the border forbidden state/s in this example?

Borderline forbidden states

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



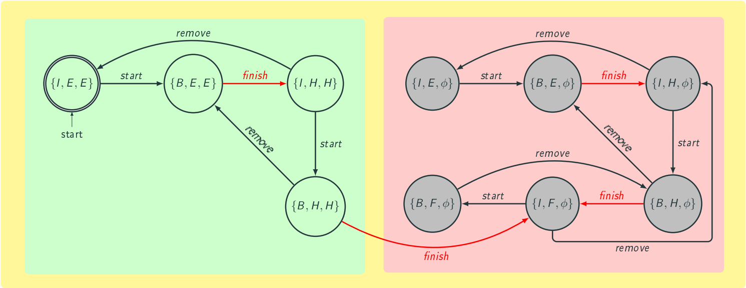
Executions of G allowed by R_1

Executions of G forbidden by R_1

$\{I, F, \phi\}$ is border forbidden.

Considerations on forbidden states

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



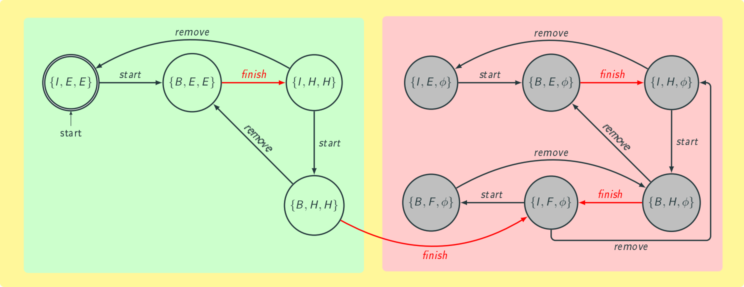
Executions of G allowed by R_1

Executions of G forbidden by R_1

- Once we enter a forbidden state, we remain in forbidden states (why?).
- What about keeping only the border forbidden ones?

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



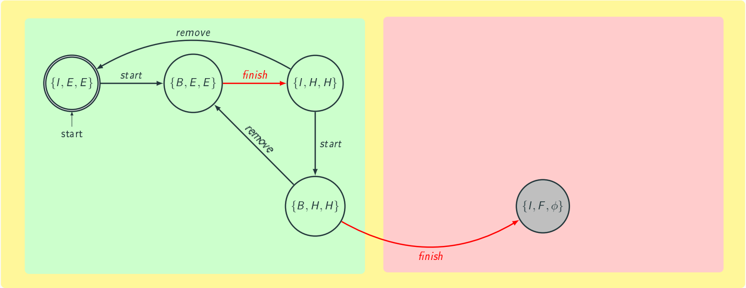
Executions of G allowed by R_1

Executions of G forbidden by R_1

Step 1: remove all forbidden states that are not border forbidden.

Supervisor Synthesis

Executions of G allowed by R'_1 (i.e., $G \| R'_1$)



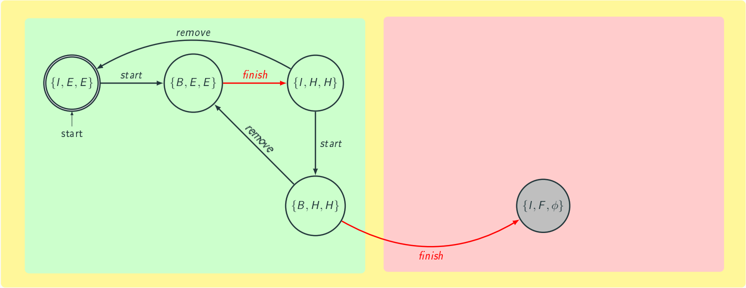
Executions of G allowed by R_1

Executions of G forbidden by R_1

Step 1: remove all forbidden states that are not border forbidden.

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



Executions of G allowed by R_1

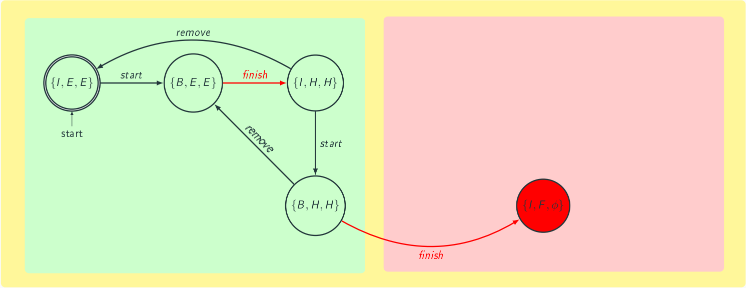
Executions of G forbidden by R_1

Step 2: $S := \mathit{Trim}'(G \| R_1')$.

Trim′ is an extension of the classic trim such that every time a transition with an uncontrollable event is removed, the trim removes also the source state of that transition (even if that state is both accessible and coaccessible).

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



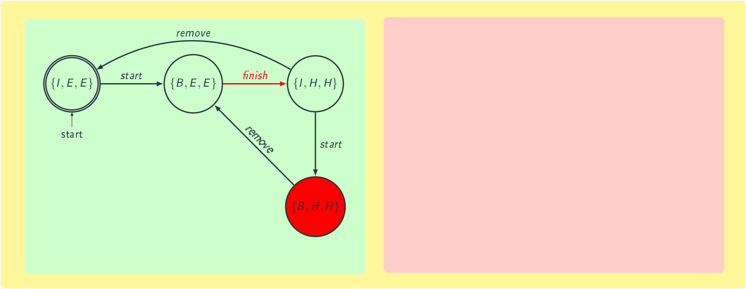
Executions of G allowed by R_1

Executions of G forbidden by R_1

- $\{I, F, \phi\}$ is non-coaccessible, thus we need to remove it.
- Watch out! The removal of $\{I, F, \phi\}$ causes the removal of *finish* which is uncontrollable. Thus, $\{B, H, H\}$ must be removed too.

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)



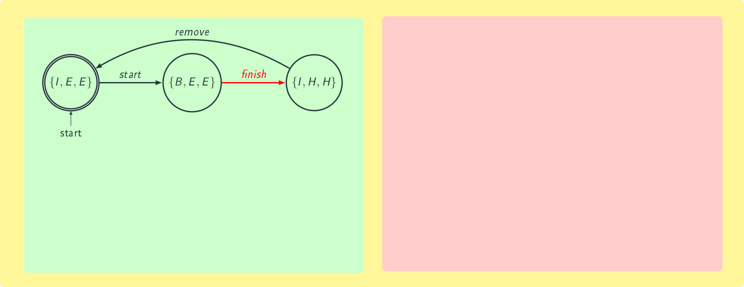
Executions of G allowed by R_1

Executions of G forbidden by R_1

- $\{B, H, H\}$ is both accessible and non-coaccessible but needs to be removed because of the removal of a blocking state.
- Notice that a controllability problem is cast as a blocking problem.
- We no longer need to reason on the original G!

Supervisor Synthesis

Executions of G allowed by R_1' (i.e., $G \| R_1'$)

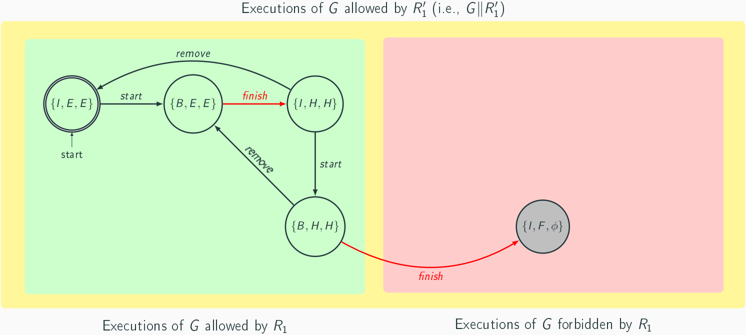


Executions of G allowed by R_1

Executions of G forbidden by R_1

Final supervisor! Looks familiar?

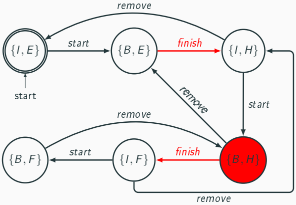
Supervisor Synthesis


Executions of G allowed by R_1' (i.e., $G \| R_1'$)
remove
start
finish
$\{I, E, E\}$
$\{B, E, E\}$
$\{I, H, H\}$
start
remove
start
$\{B, H, H\}$
$\{I, F, \phi\}$
finish
Executions of G allowed by R_1
Executions of G forbidden by R_1

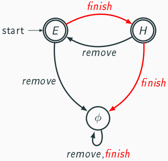
Can we improve R_1' so as to generate directly this $G \| R_1'$?

Current R_1'

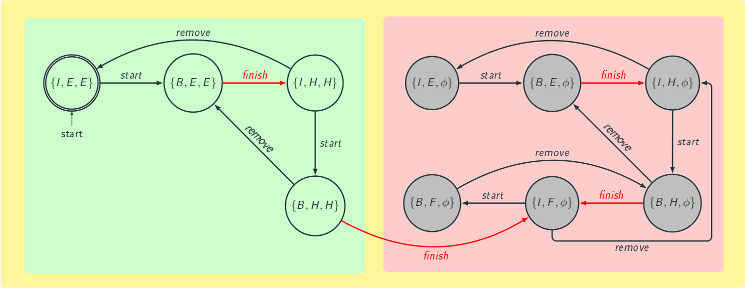
Plant G



Requirement R_1'



Executions of G allowed by R_1' (i.e., $G \| R_1'$)

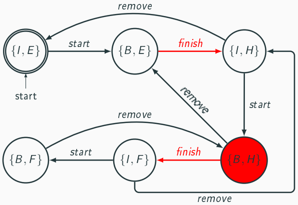


Executions of G allowed by R_1

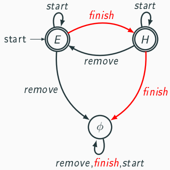
Executions of G forbidden by R_1

Improvement 1: Add self-loops for all missing events

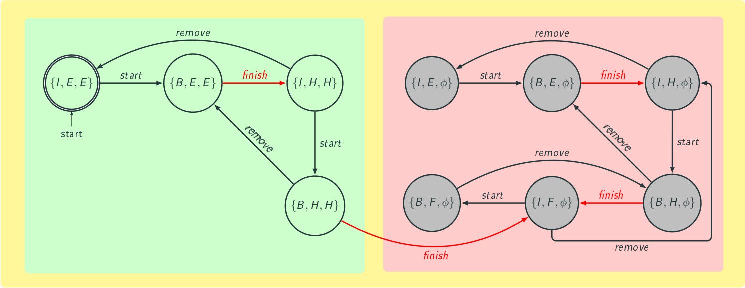
Plant G



Requirement R_1'



Executions of G allowed by R_1' (i.e., $G \| R_1'$)

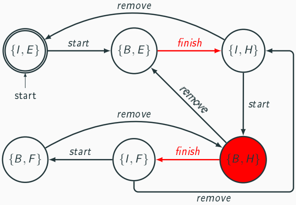


Executions of G allowed by R_1

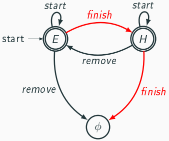
Executions of G forbidden by R_1

Remove all self-loops at ϕ

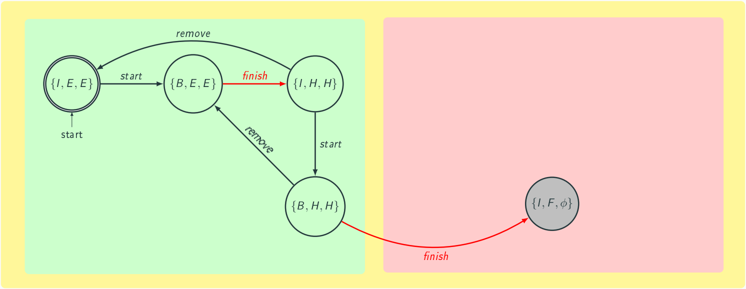
Plant G



Requirement R_1'



Executions of G allowed by R_1' (i.e., $G \| R_1'$)

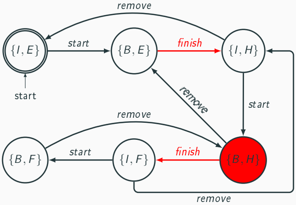


Executions of G allowed by R_1

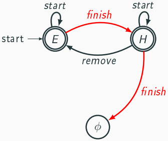
Executions of G forbidden by R_1

Remove all controllable transition leading to ϕ

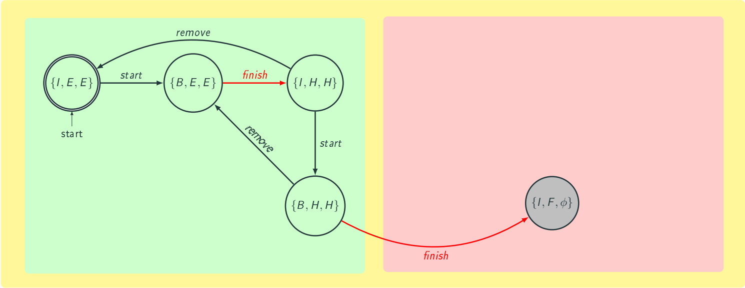
Plant G



Requirement R_1'



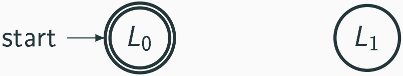
Executions of G allowed by R_1' (i.e., $G \| R_1'$)



Executions of G allowed by R_1

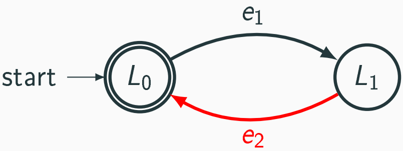
Executions of G forbidden by R_1

Extended Finite Automata: Locations and marking



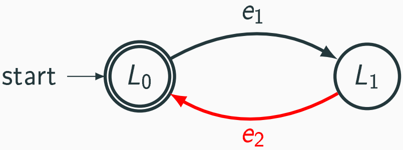
The nodes of the graph representation.

Extended Finite Automata: (Un)Controllable events



Events still label transitions.

Extended Finite Automata: Variables



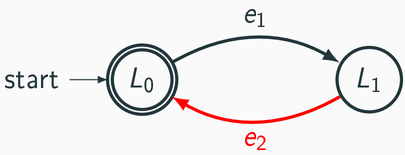
There is an underlying layer of discrete variables.

- Each variable x has a finite domain $D(x)$.
- Each variable x has an initialization value $I(x) \in D(x)$.

Here, $x \in D(x) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $I(x) := 0$.

We show **x** in a minute.

CIF basics: Discrete variables with finite domain



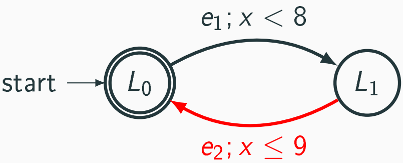
```
controllable e1;
uncontrollable e2;

plant G:
  disc int[0..9] x = 0;
  location L0: initial; marked;
    ...

  location L1:
    ...
end
```

- A discrete variable is specified by the keyword “`disc`” followed by:
  - its type;
  - its range of values (if we want its domain to be finite);
  - its initialization (default 0 for integers).

More on initialization and types: https://www.eclipse.org/escet/cif/language-tutorial/data/discvar-init.html

CIF basics: Transition guards

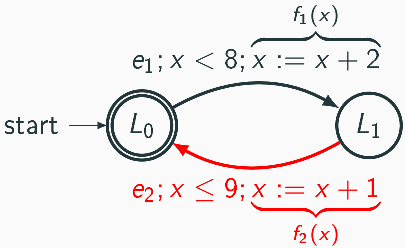
Transition guards are predicates over the variables.



```
controllable e1;
uncontrollable e2;

plant G:
  disc int[0..9] x = 0;
  location L0: initial; marked;
    edge e1 when x < 8 goto L1;

  location L1:
    edge e2 when x <= 9 goto L0;
end
```

- A guard is specified by the keyword “`when`”.

Extended Finite Automata: Transition updates

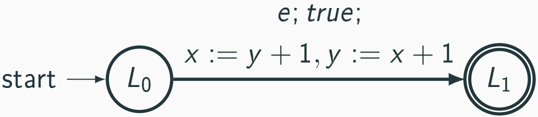

$e_1; x < 8; \overbrace{x := x + 2}^{f_1(x)}$
start
L_0
L_1
$e_2; x \leq 9; \underbrace{x := x + 1}_{f_2(x)}$

- Transition updates are functions of the variables guaranteeing that the new value of each variable x remains in $D(x)$.
- E.g., assuming that the current value of $x \in D(x) := \{0, \dots, 9\}$

$$f_1(x) := (x + 2) \mod 10$$
$$f_2(x) := (x + 1) \mod 10$$

guarantee that the new value of x remains in $D(x) := \{0, \dots, 9\}$ (similar to the overflow semantics of unsigned integers in C/C++).

Extended Finite Automata: Transition updates

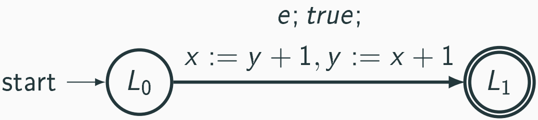


Suppose $D(x) = D(y) = \{0, \ldots, 9\}$ and $I(x) = I(y) = 1$.

Question: what are the values of x and y after executing the transition?

1) $x = 2$ and $y = 3$
2) $x = 3$ and $y = 2$
3) $x = 2$ and $y = 2$

Extended Finite Automata: Transition updates



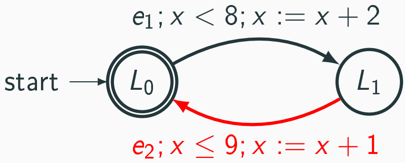
Suppose $D(x) = D(y) = \{0, \ldots, 9\}$ and $I(x) = I(y) = 1$.

Question: what are the values of x and y after executing the transition?

1) ~~$x = 2$ and $y = 3$~~
2) ~~$x = 3$ and $y = 2$~~
3) $x = 2$ and $y = 2$

- Updates are **not sensitive to the order** in which we execute the statements.
- $x := y + 1, y := x + 1$ is **equivalent to** $y := x + 1, x := y + 1$
- What really happens is $x := y' + 1$ and $y := x' + 1$, where x' and y' are the values of x and y **before executing the transition**.

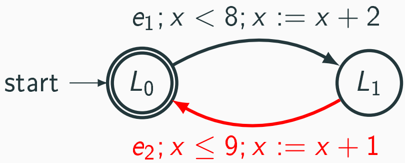
CIF basics: Transition updates



```
controllable e1;
uncontrollable e2;

plant G:
  disc int[0..9] x = 0;
  location L0: initial; marked;
    edge e1 when x < 8 do x := x + 2 goto L1;

  location L1:
    edge e2 when x <= 9 do x := x + 1 goto L0;
end
```

- An update is specified by the keyword “do”.

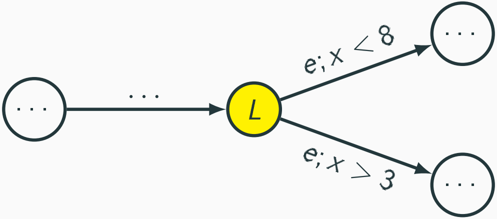
Extended Finite Automata: Transition execution



- Transition guards are predicates over the variables
- A transition (no matter if the labeling event is controllable or uncontrollable) can be executed from a location L if:
  1. the current location is L;
  2. the current value of the variables satisfies the guard.

For example, if in L_1 we have that $x = 10$, then the uncontrollable transition labeled by e_2 cannot be executed.

Extended Finite Automata: Non-determinism



In general, two transitions are non deterministic if:

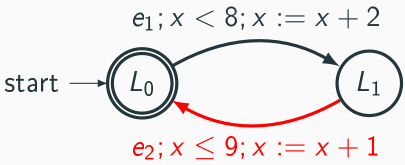
1. they are labeled by the same event;
2. the intersection of their guards is non empty.

Note that non determinism might not actually exist if the values of the variables exclude it. Suppose $D(x) := \{0, \dots, 9\}$ and that the current location is L. We have three cases:

1. if $x \leq 3$, then only the transition above can be executed;
2. if $x \geq 8$, then only the transition below can be executed;
3. if $3 < x < 8$, then both transitions can be executed.

Extended Finite Automata: States

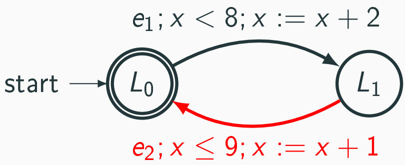
State = (Location, values of the variables)



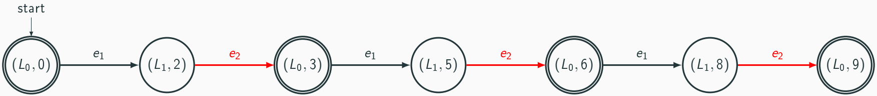
E.g., if $I(x) = 0$, then at the beginning the initial state is $(L_0, 0)$.

Extended Finite Automata: Expressiveness

$$State = (L, x)$$



Extended Finite Automata have the same expressive power of Finite State Automata. Indeed, every Extended Finite Automata can be easily encoded into a Finite State Automata. For our example:



In ESCET see `CIF miscellaneous tools -> Explore untimed state space`

Assumption regarding variables

Like events, a variable x may appear in different automata provided it complies with the following “local write/global read” contract.

Local write: x is written by one and only one automaton only;
Global read: x can be read by all automata.

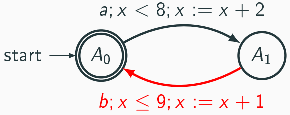
This way, we avoid

1. mismatching domains for the same variable in different automata;
2. mismatching initial values for the same variable in different automata;
3. transitions that due to synchronization write conflicting values for the same variable.

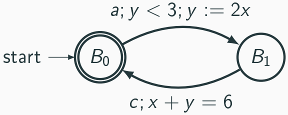
In other words, it is a form of “concurrency safety”.

Parallel composition of extended finite automata

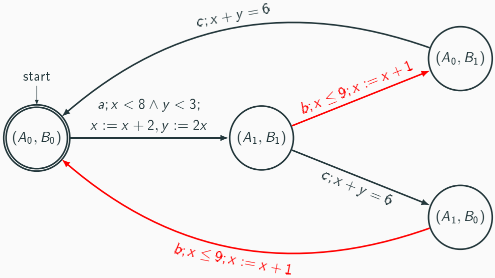
Automaton A



Automaton B



Automaton $A \| B$

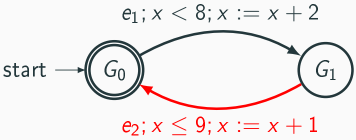


When synchronizing over the same events, the parallel composition:

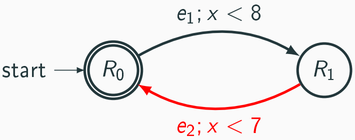
1. conjuncts the guards
2. joins the updates

Supervisory control of extended finite automata

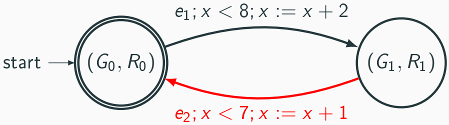
Plant G



Requirement R



Automaton $G \| R$

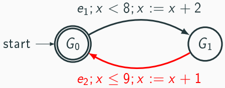


Suppose that $D(x) := \{0, \ldots, 9\}$ and $I(x) = 0$.

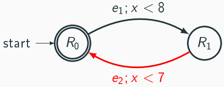
Can you spot any problem?

Supervisory control of extended finite automata

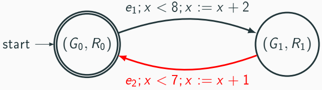
Plant G

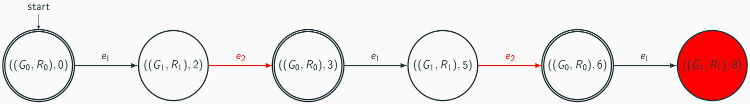


Requirement R



Automaton $G \| R$



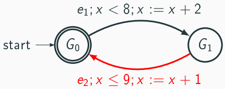


Problem: Consider the state $((G_1, R_1), 8)$ of $G \| R$. Then,

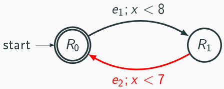
- the plant G is in state $(G_1, 8)$ and in that state G can actually take the uncontrollable transition labeled by e_2 since $x \leq 9$;
- the requirement R is in state $(R_1, 8)$ and disables the transition labeled by e_2 since R requires that $x < 7$, which is not. But e_2 is uncontrollable, so R can't actually do that.

Supervisory control of extended finite automata

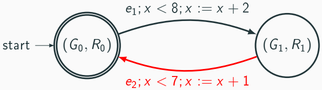
Plant G



Requirement R



Automaton $G \| R$



In supervisory control of extended finite automata:

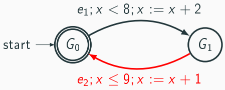
1. we do not explode the original extended finite automata into finite state automata;
2. we work **symbolically** by tightening the guards of the controllable transitions of the initial supervisor rather than removing locations.

Supervisory control of extended finite automata

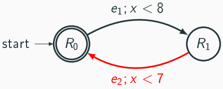
However, we need to keep track of all executions that are:

1. **possible** in the plant;
2. **forbidden** by the requirement.

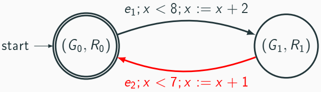
Plant G



Requirement R



Automaton $G \| R$



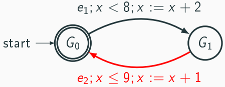
Is $G \| R$ OK for this purpose?

Supervisory control of extended finite automata

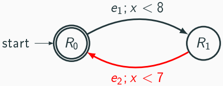
We need to keep track of all executions that are:

1. **possible** in the plant;
2. **forbidden** by the requirement.

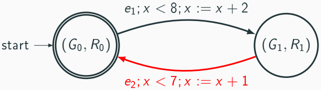
Plant G



Requirement R



Automaton $G \| R$



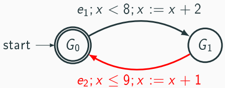
Is $G \| R$ OK for this purpose?

No! It totally misses all executions of G that are forbidden by R.

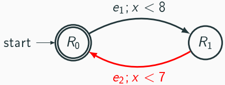
However, we know how to rewrite R into an R' so that all forbidden executions of the plant are kept in $G \| R'$, don't we?

Forbidden Plant Executions

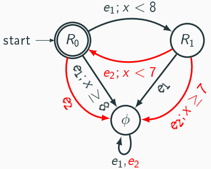
Plant G



Requirement R

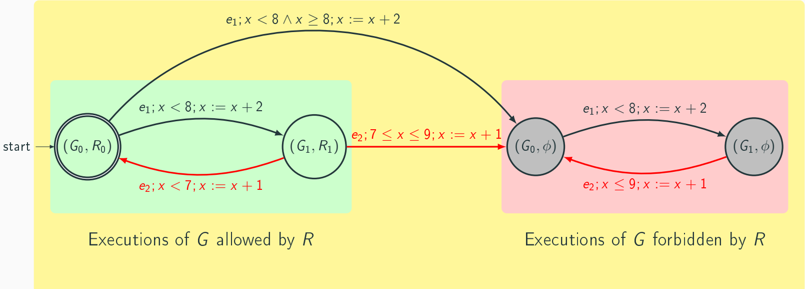


Requirement R'



Automaton $G \| R'$

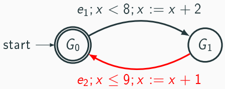
Executions of G allowed by R' (i.e., $G \| R'$)



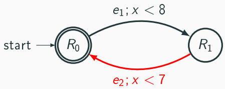
$e_1; x < 8 \wedge x \geq 8; x := x + 2$
start
(G_0, R_0)
(G_1, R_1)
$e_1; x < 8; x := x + 2$
$e_2; x < 7; x := x + 1$
$e_2; 7 \leq x \leq 9; x := x + 1$
(G_0, ϕ)
(G_1, ϕ)
$e_1; x < 8; x := x + 2$
$e_2; x \leq 9; x := x + 1$
Executions of G allowed by R
Executions of G forbidden by R

Essential R' - Keep Border Forbidden State Only

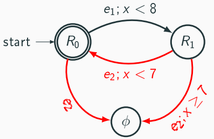
Plant G



Requirement R

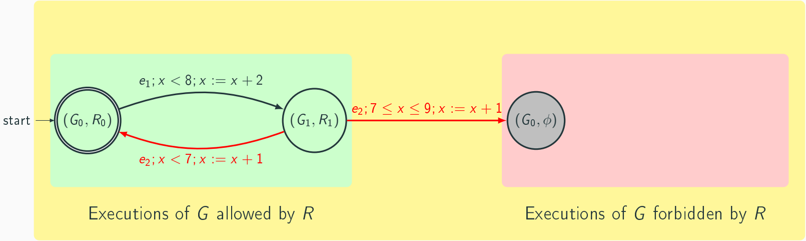


Requirement R'



Automaton $G \| R'$

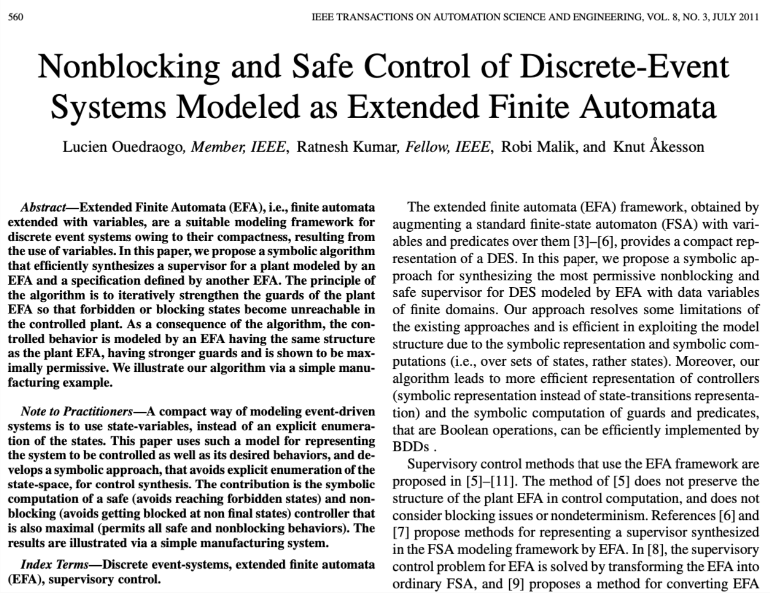
Executions of G allowed by R' (i.e., $G \| R'$)



start
(G_0, R_0)
$e_1; x < 8; x := x + 2$
(G_1, R_1)
$e_2; x < 7; x := x + 1$
$e_2; 7 \leq x \leq 9; x := x + 1$
(G_0, ϕ)
Executions of G allowed by R
Executions of G forbidden by R

- Now we can work at plant level
- Controllability problems will be modeled as blocking problems

Supervisory control of extended finite automata



Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata

Lucien Ouedraogo, *Member, IEEE*, Ratnesh Kumar, *Fellow, IEEE*, Robi Malik, and Knut Åkesson

***Abstract*—Extended Finite Automata (EFA), i.e., finite automata extended with variables, are a suitable modeling framework for discrete event systems owing to their compactness, resulting from the use of variables. In this paper, we propose a symbolic algorithm that efficiently synthesizes a supervisor for a plant modeled by an EFA and a specification defined by another EFA. The principle of the algorithm is to iteratively strengthen the guards of the plant EFA so that forbidden or blocking states become unreachable in the controlled plant. As a consequence of the algorithm, the controlled behavior is modeled by an EFA having the same structure as the plant EFA, having stronger guards and is shown to be maximally permissive. We illustrate our algorithm via a simple manufacturing example.**

***Note to Practitioners*—A compact way of modeling event-driven systems is to use state-variables, instead of an explicit enumeration of the states. This paper uses such a model for representing the system to be controlled as well as its desired behaviors, and develops a symbolic approach, that avoids explicit enumeration of the state-space, for control synthesis. The contribution is the symbolic computation of a safe (avoids reaching forbidden states) and nonblocking (avoids getting blocked at non final states) controller that is also maximal (permits all safe and nonblocking behaviors). The results are illustrated via a simple manufacturing system.**

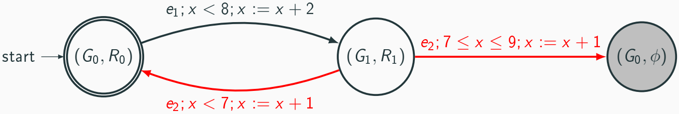
***Index Terms*—Discrete event-systems, extended finite automata (EFA), supervisory control.**

The extended finite automata (EFA) framework, obtained by augmenting a standard finite-state automaton (FSA) with variables and predicates over them [3]–[6], provides a compact representation of a DES. In this paper, we propose a symbolic approach for synthesizing the most permissive nonblocking and safe supervisor for DES modeled by EFA with data variables of finite domains. Our approach resolves some limitations of the existing approaches and is efficient in exploiting the model structure due to the symbolic representation and symbolic computations (i.e., over sets of states, rather states). Moreover, our algorithm leads to more efficient representation of controllers (symbolic representation instead of state-transitions representation) and the symbolic computation of guards and predicates, that are Boolean operations, can be efficiently implemented by BDDs .

Supervisory control methods that use the EFA framework are proposed in [5]–[11]. The method of [5] does not preserve the structure of the plant EFA in control computation, and does not consider blocking issues or nondeterminism. References [6] and [7] propose methods for representing a supervisor synthesized in the FSA modeling framework by EFA. In [8], the supervisory control problem for EFA is solved by transforming the EFA into ordinary FSA, and [9] proposes a method for converting EFA

Supervisory control of extended finite automata - algorithm

We start from the parallel composition $G \| R'$, where R' is augmented to keep track of forbidden plant executions.



After that we repeat the following three steps until fixpoint:

1. compute the **non-blocking conditions**;
2. compute the **bad state conditions**;
3. **tighten guards** of transitions with controllable events only.

The resulting extended finite automaton is the supervisor if and only if the initial state is not bad (we'll see later).

Before we start

In the following, we will often use this notation

$$P[u]$$

The meaning of this notation is a predicate obtained by P in which all occurrences of the variables of P are replaced by the right-hand sides of their updates in u.

To give some examples:

- $x = 3[x := 5]$ becomes $5 = 3$ and thus *false*;
- $x > 7[x := y + 1]$ becomes $y + 1 > 7$ and thus to $y > 6$;
- $x > y - 3[x := y - x, y := 2]$ becomes $y - x > 2 - 3$ and thus $y - x > -1$;
- $x + y = z[x := y, y := x, z := x + y]$ becomes $y + x = x + y$ and thus *true*.

Non-blocking conditions

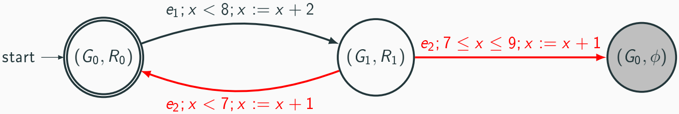
- The first phase of the algorithm requires to compute for each location of $G\|R'$ a predicate that states for which values of the variables the location is nonblocking.
- This is done iteratively until such predicates no longer change.

The concrete operations are the following.

- Initialization: $N_L := \begin{cases} \mathit{true} & \text{if } L \text{ is a marked location} \\ \mathit{false} & \text{otherwise} \end{cases}$;
- Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$.

Step 1 - Non-blocking conditions - Iteration 1

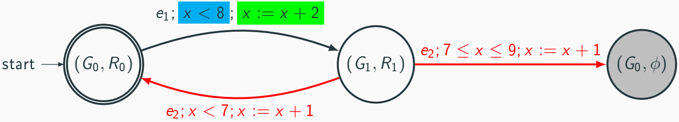
$$\text{Initialization: } N_L := \begin{cases} \mathit{true} & \text{if } L \text{ is a marked location} \\ \mathit{false} & \text{otherwise} \end{cases}$$



Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*

Step 1 - Non-blocking conditions - Iteration 2

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$

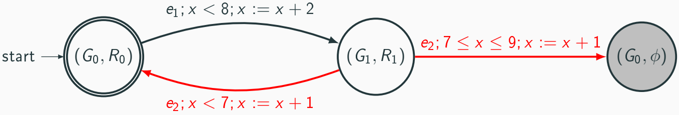


Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*		

$$N_{(G_0,R_0)} := true \vee (x < 8 \wedge false[x := x + 2])$$
$$= true \vee (x < 8 \wedge false)$$
$$= true$$

Step 1 - Non-blocking conditions - Iteration 2

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$

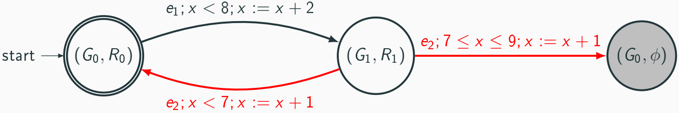


Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*	x < 7	

$$\begin{aligned} N_{(G_1,R_1)} &:= \mathit{false} \vee ((x < 7 \wedge \mathit{true}[x := x + 1]) \vee (7 \leq x \leq 9 \wedge \mathit{false}[x := x + 1])) \\ &= \mathit{false} \vee (x < 7 \vee \mathit{false}) \\ &= \mathit{false} \vee x < 7 \\ &= x < 7 \end{aligned}$$

Step 1 - Non-blocking conditions - Iteration 2

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$

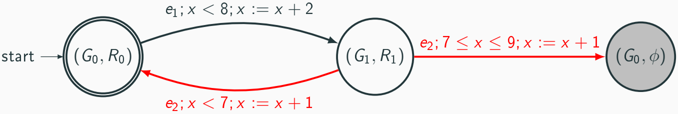


Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*	$x < 7$	*false*

$$N_{(G_0,\phi)} := \mathit{false}$$

Step 1 - Non-blocking conditions - Iteration 3

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$



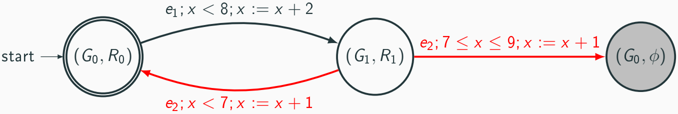
Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*	$x < 7$	*false*
3	*true*		

$$N_{(G_0,R_0)} := true \vee (x < 8 \wedge x < 7[x := x + 2])$$

$$= true$$

Step 1 - Non-blocking conditions - Iteration 3

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$



Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*	$x < 7$	*false*
3	*true*	$x < 7$	

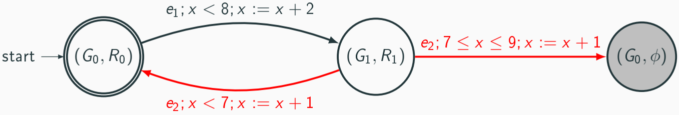
$$N_{(G_1,R_1)} := x < 7 \vee ((x < 7 \wedge true[x := x + 1]) \vee (7 \leq x \leq 9 \wedge false[x := x + 1]))$$

$$= x < 7 \vee (x < 7 \vee false)$$

$$= x < 7$$

Step 1 - Non-blocking conditions - Iteration 3

Update: $N_L := N_L \vee (\bigvee_{L \xrightarrow{e;g;u} L'} (g \wedge N_{L'}[u]))$



Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
1	*true*	*false*	*false*
2	*true*	$x < 7$	*false*
3	*true*	$x < 7$	*false*

$$N_{(G_0,\phi)} := \mathit{false}$$

We reached a fixpoint so we are done with this step for the moment.

Bad state conditions

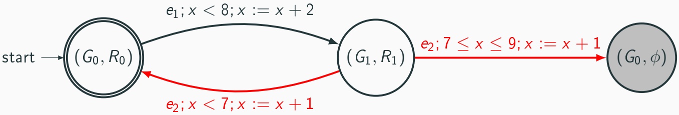
- The synthesis algorithm must not restrict uncontrollable events
- Restrictions on uncontrollable events are propagated backwards until an edge with a controllable event is encountered.
- This is achieved by the bad state condition computation.
- We compute a bad state condition for each location.
- This is done iteratively until such predicates no longer change.

The concrete operations are the following.

- Initialization: $B_L := \neg N_L$
- Update: $B_L := B_L \vee (\bigvee_{L \xrightarrow{e;g;u} L', e \in E_u} (g \wedge B_{L'}[u]))$ where e **is an uncontrollable event**.

Step 2 - Bad state conditions - Iteration 1

Initialization: $B_L := \neg N_L$



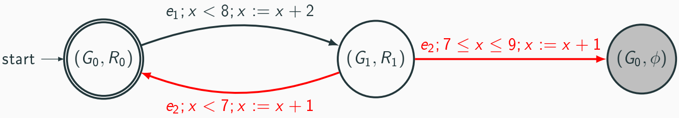
Iteration	$N_{(G_0,R_0)}$	$N_{(G_1,R_1)}$	$N_{(G_0,\phi)}$
...	...	...	...
3	*true*	$x < 7$	*false*

↓

Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
1	*false*	$x \geq 7$	*true*

Step 2 - Bad state conditions - Iteration 2

Update: $B_L := B_L \vee (\bigvee_{L \xrightarrow{e;g;u} L', e \in E_u} (g \wedge B_{L'}[u]))$

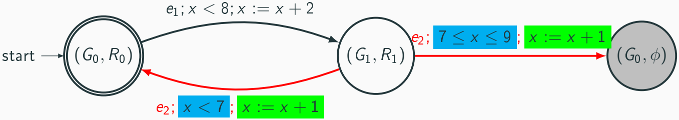


Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
1	*false*	$x \geq 7$	*true*
2	*false*		

$$B_{(G_0,R_0)} := \mathit{false}$$

Step 2 - Bad state conditions - Iteration 2

Update: $B_L := B_L \vee (\bigvee_{L \xrightarrow{e;g;u} L', e \in E_u} (g \wedge B_{L'}[u]))$



Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
1	*false*	$x \geq 7$	*true*
2	*false*	$x \geq 7$	

$$B_{(G_1,R_1)} := x \geq 7 \vee ((x < 7 \wedge false[x := x + 1]) \vee (7 \leq x \leq 9 \wedge true[x := x + 1]))$$

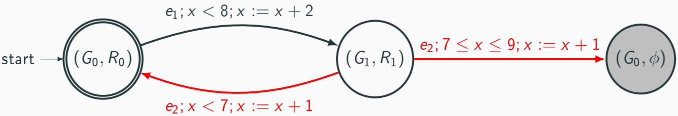
$$= x \geq 7 \vee (false \vee 7 \leq x \leq 9)$$

$$= x \geq 7 \vee (7 \leq x \leq 9)$$

$$= x \geq 7$$

Step 2 - Bad state conditions - Iteration 2

Update: $B_L := B_L \vee (\bigvee_{L \xrightarrow{e;g;u} L', e \in E_u} (g \wedge B_{L'}[u]))$



Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
1	*false*	$x \geq 7$	*true*
2	*false*	$x \geq 7$	*true*

$$B_{(G_0,\phi)} := true$$

Tightening of controllable guards

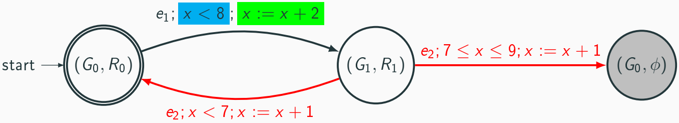
- Bad state conditions express which combinations of values of variables need to be avoided in a specific location, considering that guards of uncontrollable events can't be touched.
- The guards of the edges with a controllable event are updated by adding as a conjunct the expression $\neg B_L[u]$ where $B_L[u]$ is the bad state condition of the target location L.

The concrete operation is the following.

- $L \xrightarrow{e;g;u} L'$ with $e \in E_c$ is tightened to $L \xrightarrow{e;g \wedge \neg B_{L'}[u];u} L'$

Step 3 - Tightening of guards labeled by controllable events

Tightening the transition labeled by e_1.



Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
...	...	...	...
2	*false*	$x \geq 7$	*true*

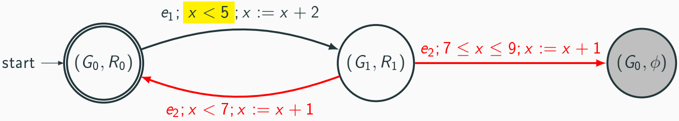
$$x < 8 \land \overbrace{\neg(x \geq 7\,[x := x + 2])}^{\neg B_{(\boldsymbol{G_1},\boldsymbol{R_1})}[x:=x+2]}$$

$$= x < 8 \land \neg(x + 2 \geq 7)$$

$$= x < 8 \land \neg(x \geq 5)$$

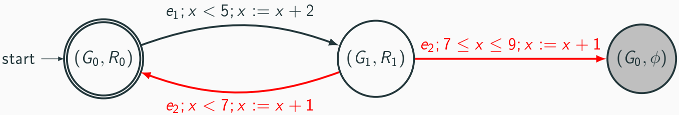
$$= x < 8 \land x < 5$$

$$= x < 5$$



Are we ready to go?

- If we iterate all three steps again (on the tightened $G \| R'$) nothing changes.
- This resulting automaton is our tentative supervisor.



Control exists if the initial conditions on the variables do not satisfy the bad location predicate of the initial location.

Iteration	$B_{(G_0,R_0)}$	$B_{(G_1,R_1)}$	$B_{(G_0,\phi)}$
...	...	...	...
2	*false*	$x \geq 7$	*true*

$$x = 0 \not\models B_{(G_0,R_0)}$$

$$x = 0 \not\models \mathit{false}$$

true

We have control!

State-based requirements

- When looking at case studies, we often observe that system requirements are naturally expressed in terms of conditions over **states**.
- Designers can express requirements more easily by using such state-based specifications because they naturally follow from informal, intuitive requirements

Two kinds of state-based requirements:

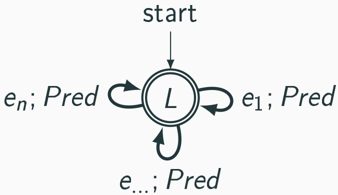
1. Event conditions;
2. Invariants.

State-based requirements: event conditions

Let $E := \{e_1, \ldots, e_n\}$ be a set of events. An event condition is an expression of the form:

$$E \Rightarrow Pred$$

meaning that the events in E can only be executed if $Pred$ is satisfied.



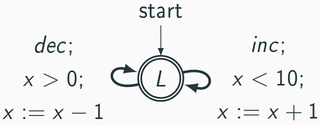
```
requirement R:
  location:
    initial;
    marked;
    edge e1,...,en when Pred;
end
```

More compactly: `requirement R: {e1,...,en} needs Pred;`

State-based requirements: event conditions

Let x be a discrete variable with domain $D(x) := \{0, \ldots, 10\}$ and initial value $I(x) := 5$

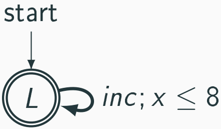
Consider the following extended finite automaton



Event condition requirements:

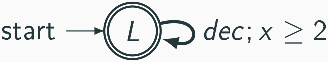
Increment is possible only if $x \leq 8$

$\{inc\} \Rightarrow x \leq 8$



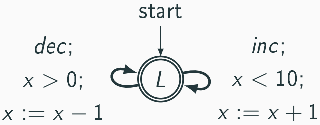
Decrement is possible only if $x \geq 2$

$\{dec\} \Rightarrow x \geq 2$



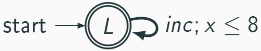
State-based requirements: event conditions

Plant:



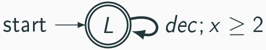
```
controllable inc, dec;
plant G:
  disc int[0..10] x = 5;
  location: initial; marked;
    edge inc when x < 10 do x := x+1;
    edge dec when x > 0  do x := x-1;
end
```

Requirement R_1: $\{inc\} \Rightarrow x \leq 8$



```
requirement R1:
  location: initial; marked;
    edge inc when G.x <= 8;
end
```

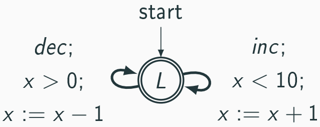
Requirement R_2: $\{dec\} \Rightarrow x \geq 2$



```
requirement R2:
  location: initial; marked;
    edge dec when G.x >= 2;
end
```

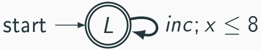
State-based requirements: event conditions

Plant:



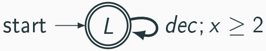
```
controllable inc, dec;
plant G:
  disc int[0..10] x = 5;
  location: initial; marked;
    edge inc when x < 10 do x := x+1;
    edge dec when x > 0  do x := x-1;
end
```

Requirement R_1: $\{inc\} \Rightarrow x \leq 8$



```
requirement R1: inc needs G.x <= 8;
```

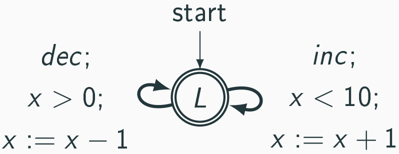
Requirement R_2: $\{dec\} \Rightarrow x \geq 2$



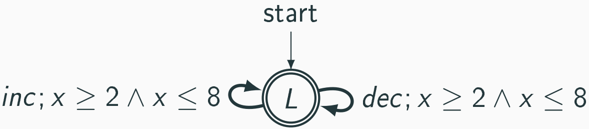
```
requirement R2: dec needs G.x >= 2;
```

State-based requirements: invariants

Plant:



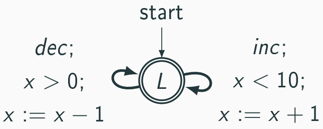
Invariant requirement: x **must *always* be between** 2 **and** 8**.**


start
L
$inc; x \geq 2 \wedge x \leq 8$
$dec; x \geq 2 \wedge x \leq 8$

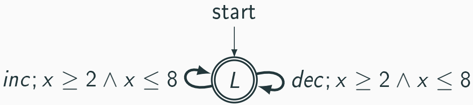
Can we use the same idea discussed before and add self-loop transitions for all edges enforcing *Pred*?

State-based requirements: invariants

Plant:



Invariant requirement: x **must** ***always*** **be between** 2 **and** 8**.**

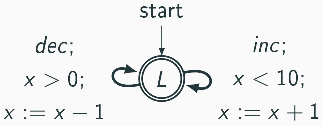

start
L
$inc; x \geq 2 \wedge x \leq 8$
$dec; x \geq 2 \wedge x \leq 8$

No! As well as holding before taking any transition, *Pred* must also hold *after* taking any transition. In this case,

- $(L,5) \xrightarrow{dec} (L,4) \xrightarrow{dec} (L,3) \xrightarrow{dec} (L,2)$ $\xrightarrow{dec} (L,1)$
- $(L,5) \xrightarrow{inc} (L,6) \xrightarrow{inc} (L,7) \xrightarrow{inc} (L,8)$ $\xrightarrow{inc} (L,9)$

State-based requirements: invariants

Plant:



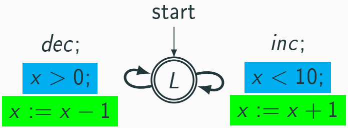
Invariant requirement: x **must *always* be between** 2 **and** 8**.**

- For each plant automaton writing a variable in *Pred* we create a requirement automaton as a copy of the original plant automaton where each transition of the original plan automaton $L \xrightarrow{e;g;u} L'$ is tightened to $L \xrightarrow{e;g \wedge Pred[u]} L'$ in the requirement automaton.

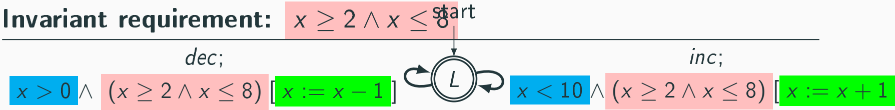
This way, if a transition is taken its guard already guarantees that *Pred* will hold after taking the transition.

State-based requirements: invariants

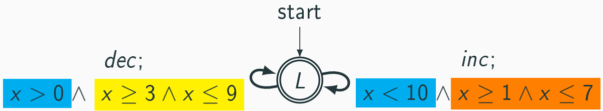
Plant:



Invariant requirement: $x \geq 2 \wedge x \leq 8$

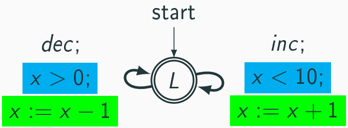

start
$dec;$
$x > 0 \wedge (x \geq 2 \wedge x \leq 8)[x := x - 1]$
L
$inc;$
$x < 10 \wedge (x \geq 2 \wedge x \leq 8)[x := x + 1$

- $(x \geq 2 \wedge x \leq 8)[x := x - 1]$ is equivalent to $(x \geq 3 \wedge x \leq 9)$
- $(x \geq 2 \wedge x \leq 8)[x := x + 1]$ is equivalent to $(x \geq 1 \wedge x \leq 7)$

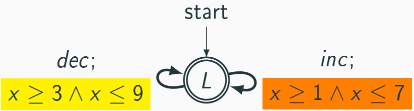

start
$dec;$
$x > 0 \wedge x \geq 3 \wedge x \leq 9$
L
$inc;$
$x < 10 \wedge x \geq 1 \wedge x \leq 7$

State-based requirements: invariants

Plant:



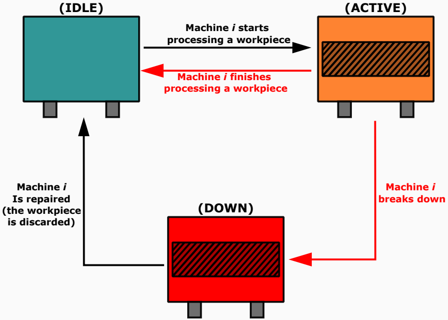
Invariant requirement: $x \geq 2 \wedge x \leq 8$

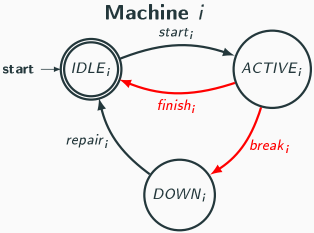


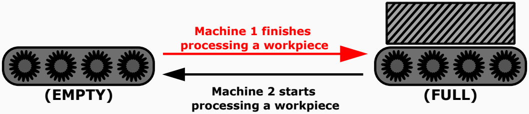
```
requirement invariant R: G.x >= 2 and G.x <= 8;
```

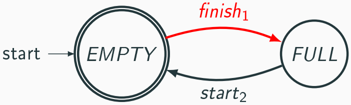
- $(L, 5) \xrightarrow{dec} (L, 4) \xrightarrow{dec} (L, 3) \xrightarrow{dec} (L, 2)$ (*dec* is disabled now)
- $(L, 5) \xrightarrow{inc} (L, 6) \xrightarrow{inc} (L, 7) \xrightarrow{inc} (L, 8)$ (*inc* is disabled now)

Locations vs Variables


(IDLE)
(ACTIVE)
Machine *i* starts processing a workpiece
Machine *i* finishes processing a workpiece
Machine *i* Is repaired (the workpiece is discarded)
Machine *i* breaks down
(DOWN)

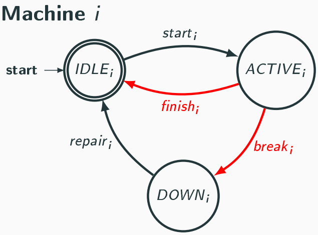


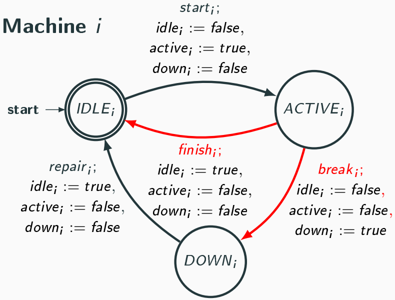

Machine 1 finishes processing a workpiece
Machine 2 starts processing a workpiece
(EMPTY)
(FULL)



Can we encode locations as variables?

Locations encoded as Variables: Machines



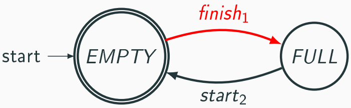


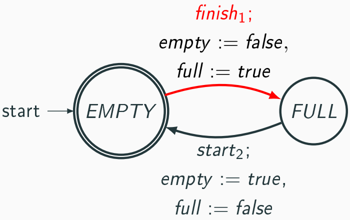
- Variables $idle_i, active_i, down_i$;
- Domains $D(idle_i) = D(active_i) = D(down_i) = \{true, false\}$;
- Initialization $I(idle_i) = true$, $I(active_i) = I(down_i) = false$;

Yes! Just add one Boolean variable l_i for each location L such that:

1. l_i is set true upon entering L;
2. l_i is set false upon leaving L.

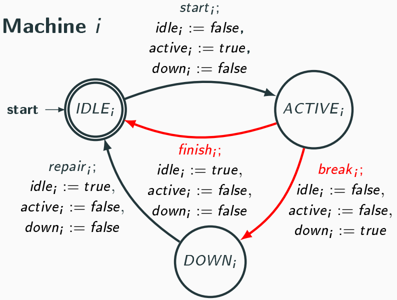
Locations encoded as Variables: Buffer

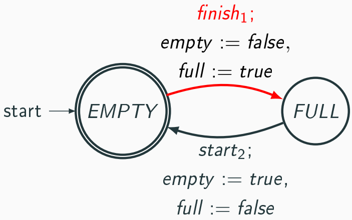




- Variables $empty, full$;
- Domains $D(empty) = D(full) = \{true, false\}$;
- Initialization $I(empty) = true$, $I(full) = false$;
- Now we can use event and invariant conditions by using location names (internally they will be replaced by the corresponding Boolean variables).
- For example, $E \Rightarrow A.L$ says that the events in E can be executed only if the automaton A is in location L.

Manufacturing process requirements





R_1: Machine 1 can start processing a workpiece only if the Buffer is empty

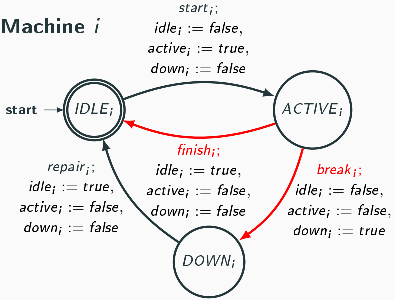
Event condition

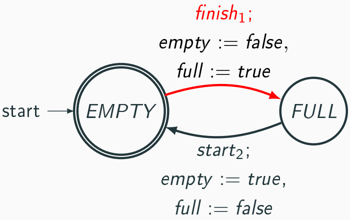
$\{start_1\} \Rightarrow B.empty$

CIF code

```
requirement R1: start1 needs B.EMPTY;
```

Manufacturing process requirements





R_2: **Machine 2 can start processing a workpiece only if the Buffer is full**

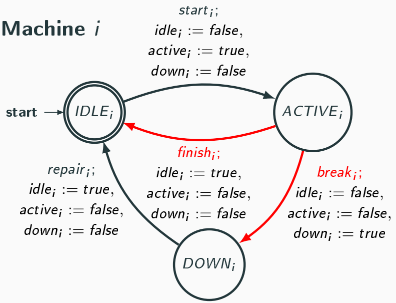
Event condition

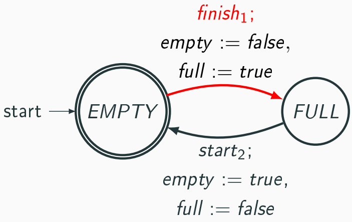
$\{start_2\} \Rightarrow B.full$

CIF code

```
requirement R2: start2 needs B.FULL;
```

Manufacturing process requirements


Machine *i*



R_3: Machine 1 cannot start processing a workpiece if Machine 2 is down.

Event condition

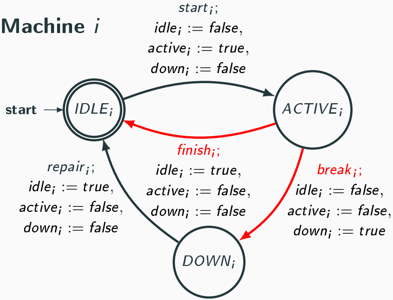
$\{start_1\} \Rightarrow \neg M_2.down$

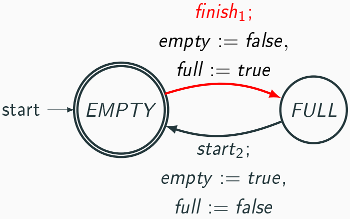
CIF code

```
requirement R3: start1 needs not M2.DOWN;
```

Manufacturing process requirements

Machine *i*





R_4: If both Machines are down, then Machine 2 is repaired before Machine 1.

Event condition

$\{repair_1\} \Rightarrow \neg M_2.down$

CIF code

```
requirement R4: repair1 needs not M2.DOWN;
```