Automated Analysis of Parametric Timing-Based Mutual Exclusion Algorithms

(slides revision: Tuesday 10th April, 2012, 21:03)

R. Bruttomesso, A. Carioni, S. Ghilardi, S. Ranise

University of Milan, Italy
FBK Trento, Italy

April 4, 2012

Outline

1. Introduction
2. Mutual Exclusion Algorithms
   - Asynchrounous Mutual Exclusion Algorithms
   - Infinite-State Safety Verification with MCMT
3. Mutual Exclusion and Deadlock Freedom
   - Timing-based Mutual Exclusion Algorithms
   - Infinite-State Safety Verification with MCMT
4. Deadlock-Freedom via Waiting time constraints
   - Verification
   - Automatic Verification
   - Automatic Synthesis
5. Conclusion

Introduction

Mutual Exclusion Algorithms

- *remainder*: the region of code not concerned with the access to critical resources
- *trying*: the region of code where the process tries to acquire access to the critical region
- *critical*: the region of code with exclusive access
- *exit*: the region of code where the process exits from the critical region

Process i:

repeat forever
  remainder region
  trying region
  critical region
  exit region
end repeat

Properties of interest

- Mutual Exclusion (MEX): in any reachable state, at most one process is in its critical region (it is a safety property)
- Deadlock Freedom (DF): in any execution, if some process is in the trying region, and no process is in the critical region, then eventually some process enters the critical region (it is not a safety property)

Process i:

repeat forever
remainder region
trying region
critical region
exit region
end repeat

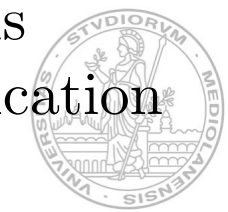
Properties of interest

- Mutual Exclusion (MEX): in any reachable state, at most one process is in its critical region (it is a safety property)
- Deadlock Freedom (DF): in any execution, if some process is in the trying region, and no process is in the critical region, then eventually some process enters the critical region (it is not a safety property)

Process i:

repeat forever
remainder region
trying region
critical region
exit region
end repeat

We assume there are N processes running concurrently (but N is **never fixed** during verification, it's a parameter: result of verification is valid **for every** N.)



Mutual Exclusion Algorithms

Lamport's Mutual Exclusion Algorithm

- Simple algorithm based on two "barriers"
- Uses two shared registers
  - x holds process id that may access the critical region
  - y set to 1 when some process wants to access the critical region

Algorithm 1

x, y: *shared registers*
initially $y = 0$

repeat forever
0: *remainder* $exit_i$
1: $x := i$;
2: **if** $y \neq 0$ **then goto** 1;
3: $y := 1$;
4: **if** $x \neq i$ **then goto** 1;
5: *critical* $entry_i$
6: *critical* $exit_i$
7: $y := 0$;
8: *remainder* $entry_i$
end repeat

Infinite-State Safety Verification with MCMT

- MCMT (Model Checking Modulo Theories)[1] verifies safety of *array-based-systems*, which are suitable to encode interactions of processes in a protocol
- Intuitively, $a[i]$ indicates a value for process i (array is not bounded)

[1] Please ref. to – *Backward Reachability of Array-based Systems by SMT-solving: Termination and Invariant Synthesis*, Ghilardi and Ranise – for a thorough and precise description.

Infinite-State Safety Verification with MCMT

- MCMT (Model Checking Modulo Theories)[1] verifies safety of *array-based-systems*, which are suitable to encode interactions of processes in a protocol
- Intuitively, $a[i]$ indicates a value for process i (array is not bounded)
- An array-based-system is a transition system

$$(\underline{a},\ I,\ \tau)$$

where

  - $\underline{a}$ is a tuple of arrays (they are the state variables)
  - I is the initial state of the form $\forall \underline{i}.\ \varphi(\underline{i}, a[\underline{i}])$
  - τ is the transition relation of the form $\bigvee_l \tau_l$ where $\tau_l = \exists \underline{i}.\ (\ \varphi(\underline{i}, a[\underline{i}])\ \wedge\ Update(\underline{i}, a', a)\ )$

[1] Please ref. to – *Backward Reachability of Array-based Systems by SMT-solving: Termination and Invariant Synthesis*, Ghilardi and Ranise – for a thorough and precise description.

Lamport's Mutual Exclusion Algorithm

- Initial I

$$\forall i.\ (\ y = 0\ \wedge\ pc[i] = 0\ )$$

- Transition τ_1

$$\exists i.\ (\quad pc[i] = 0\ \wedge \\ pc' = \lambda j.\ \mathrm{ite}(i = j,\ 1,\ pc[j])\ )$$

- Transition Relation $T = \bigvee_l \tau_l$

Algorithm 1

x, y: shared registers
initially $y = 0$

repeat forever
0: *remainder* $exit_i$
1: $x := i$;
2: **if** $y \neq 0$ **then goto** 1;
3: $y := 1$;
4: **if** $x \neq i$ **then goto** 1;
5: *critical* $entry_i$
6: *critical* $exit_i$
7: $y := 0$;
8: *remainder* $entry_i$
end repeat

Lamport's Mutual Exclusion Algorithm

- Initial I

$$\forall i.\ (\ y = 0\ \wedge\ pc[i] = 0\ )$$

- Transition τ_1

$$\exists i.\ (\quad pc[i] = 0\ \wedge \\ pc' = \lambda j.\ \mathrm{ite}(i = j,\ 1,\ pc[j])\ )$$

- Transition Relation $T = \bigvee_l \tau_l$
- (Negation of) Property MEX

$$\exists i, k.\ (\qquad i \neq k \qquad \wedge \\ 5 \leq pc[i] \leq 6 \quad \wedge \\ 5 \leq pc[k] \leq 6 \quad)$$

Algorithm 1

x, y: shared registers
initially $y = 0$

repeat forever
0: *remainder* $exit_i$
1: $x := i$;
2: **if** $y \neq 0$ **then goto** 1;
3: $y := 1$;
4: **if** $x \neq i$ **then goto** 1;
5: *critical* $entry_i$
6: *critical* $exit_i$
7: $y := 0$;
8: *remainder* $entry_i$
end repeat

Infinite-State Safety Verification with MCMT

MCMT explores the state space in a **backward** manner, i.e., it explores the space of **unsafe** states starting from the negation of the property

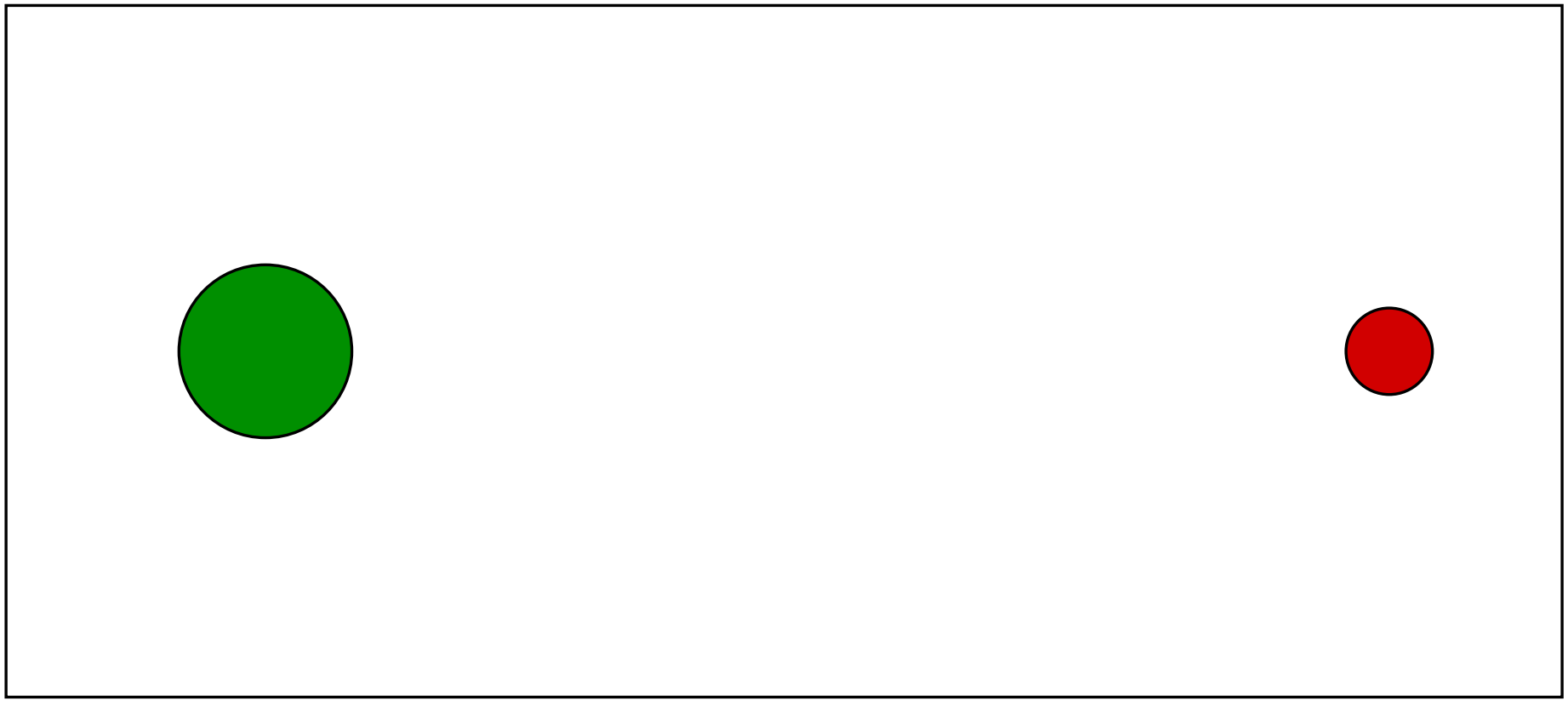
Preimage computation is done in such a way that (previous) unsafe states remain in the form $\exists \underline{i}.\ \varphi(\underline{i}, \underline{a}[i])$

MCMT leverage on the efficiency of SMT-solvers to execute two checks:

- **Safety check**: checks intersection of an unsafe state and I.
  If `SAT` then property violated (system is unsafe)
- **Fix-point check**: check if unsafe states are inductive w.r.t. τ.
  If `VALID` (and Safety check `UNSAT`) then property holds (system is safe)

Infinite-State Safety Verification with MCMT

System Status: `UNKNOWN`



 Initial States

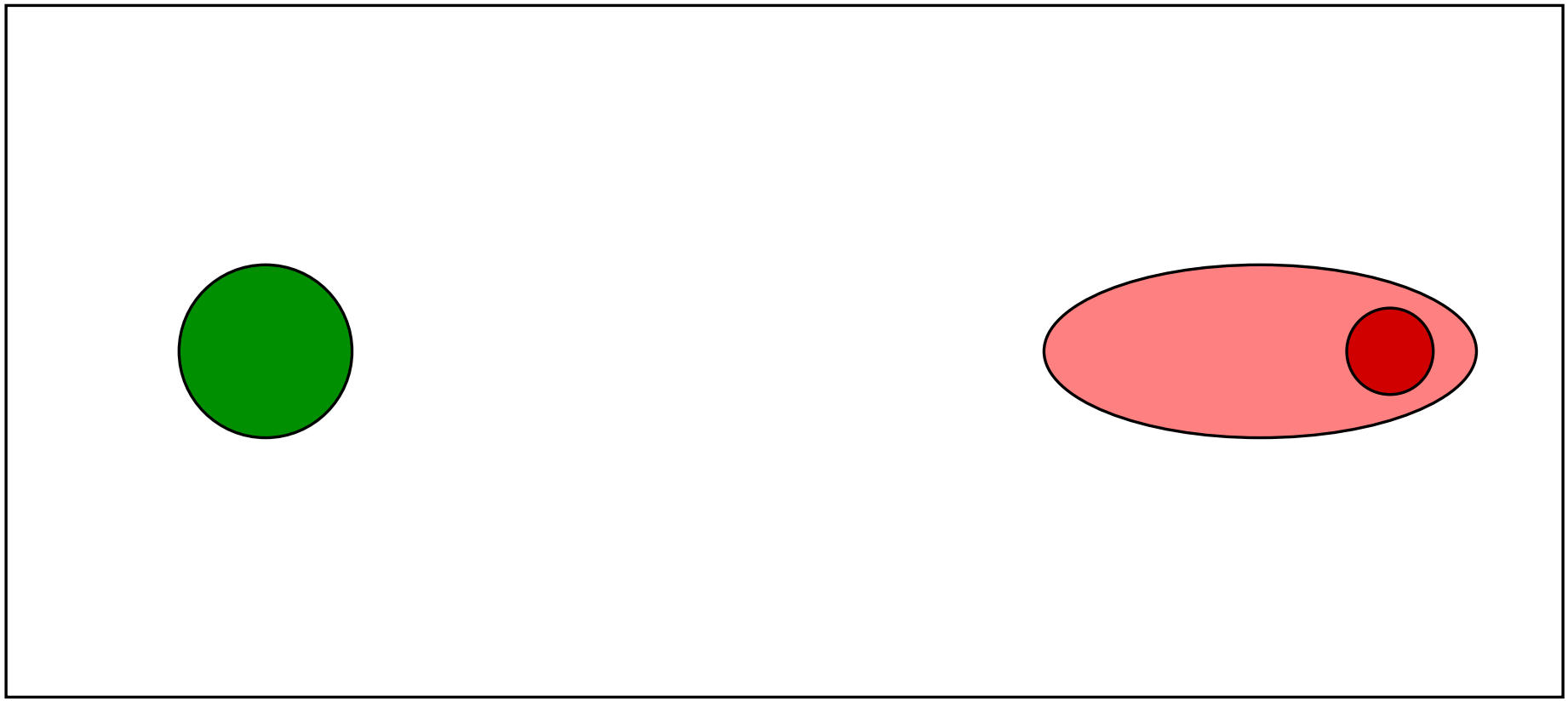
 Unsafe/Bad States

 Reachable States

 Reachable Bad States

Infinite-State Safety Verification with MCMT

System Status: `UNKNOWN`



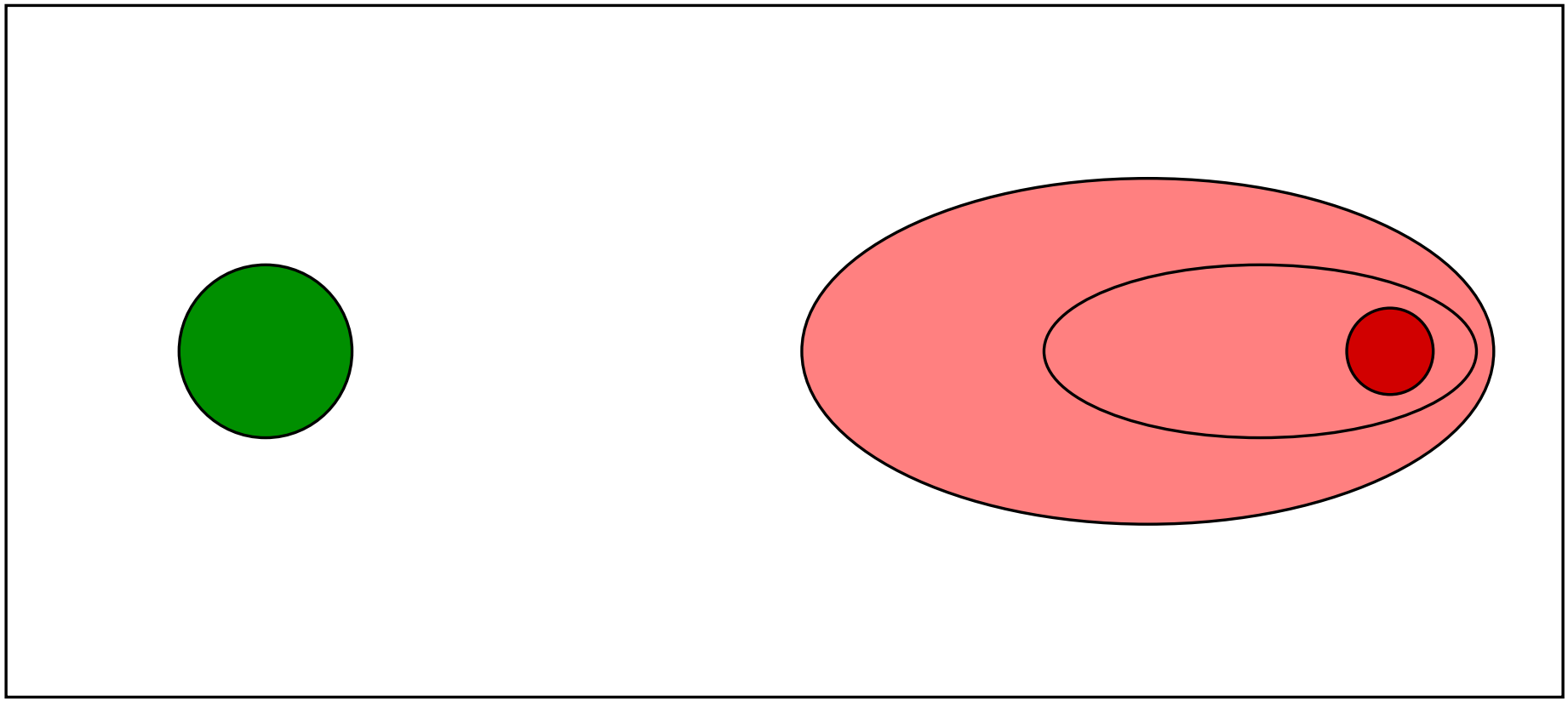
 Initial States

 Unsafe/Bad States

 Reachable States

 Reachable Bad States

System Status: `UNKNOWN`



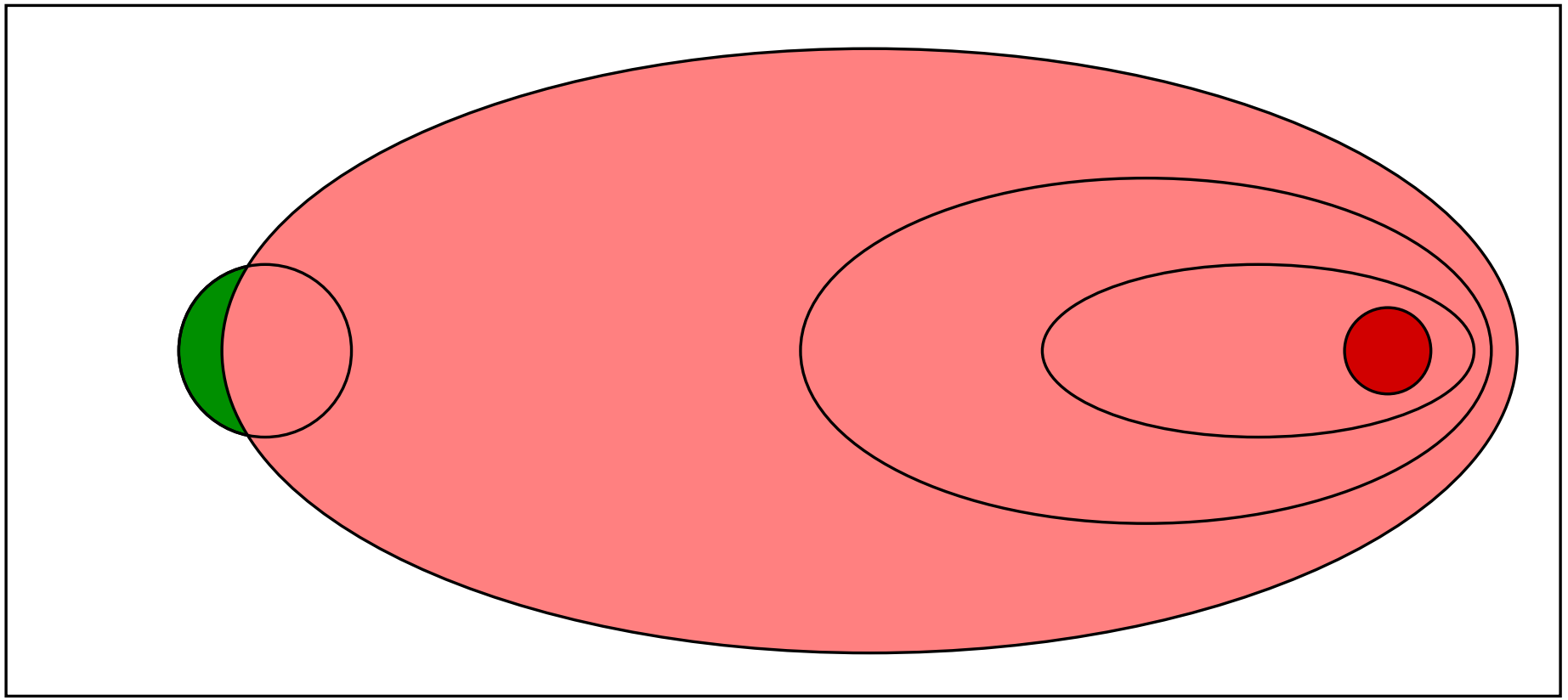
Initial States

Unsafe/Bad States

Reachable States

Reachable Bad States

System Status: `UNSAFE`



Initial States

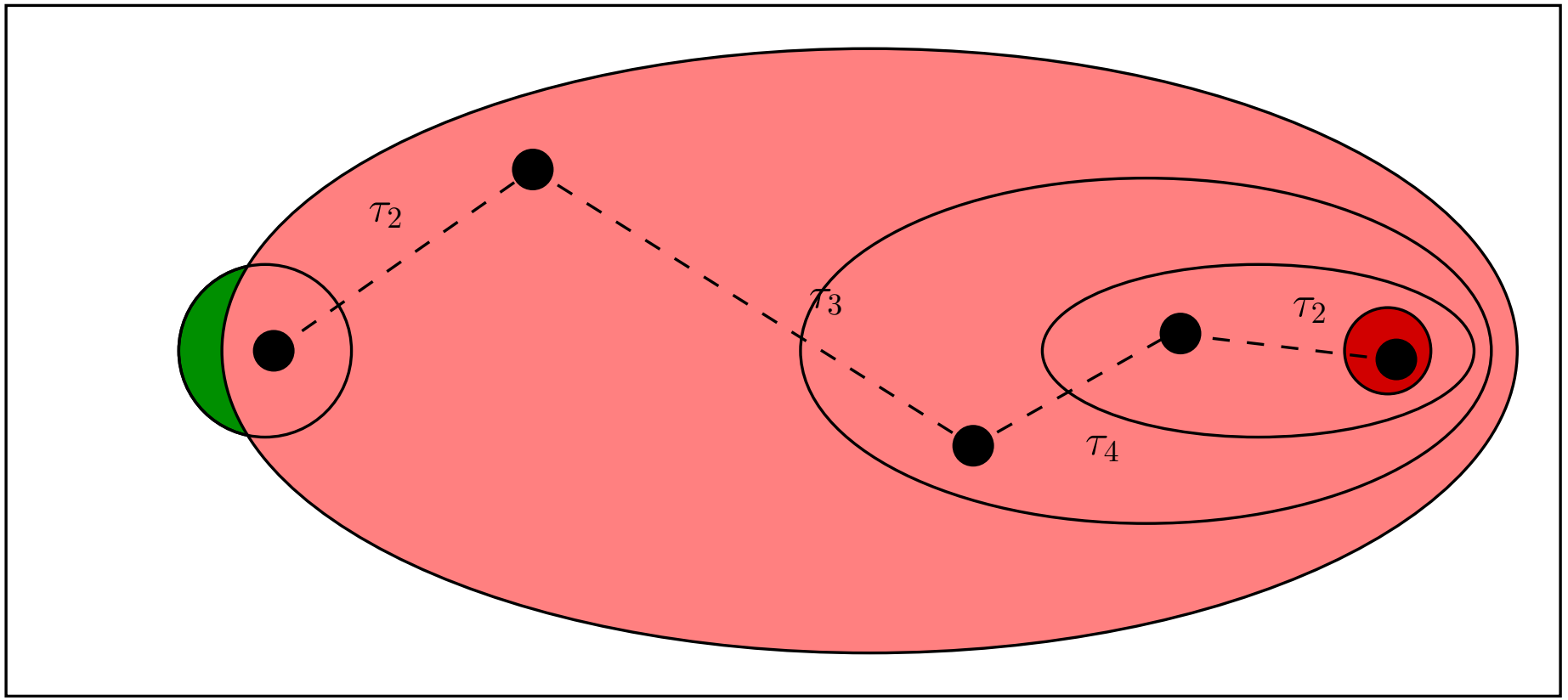
Reachable States

Unsafe/Bad States

Reachable Bad States

Infinite-State Safety Verification with MCMT

System Status: `UNSAFE`



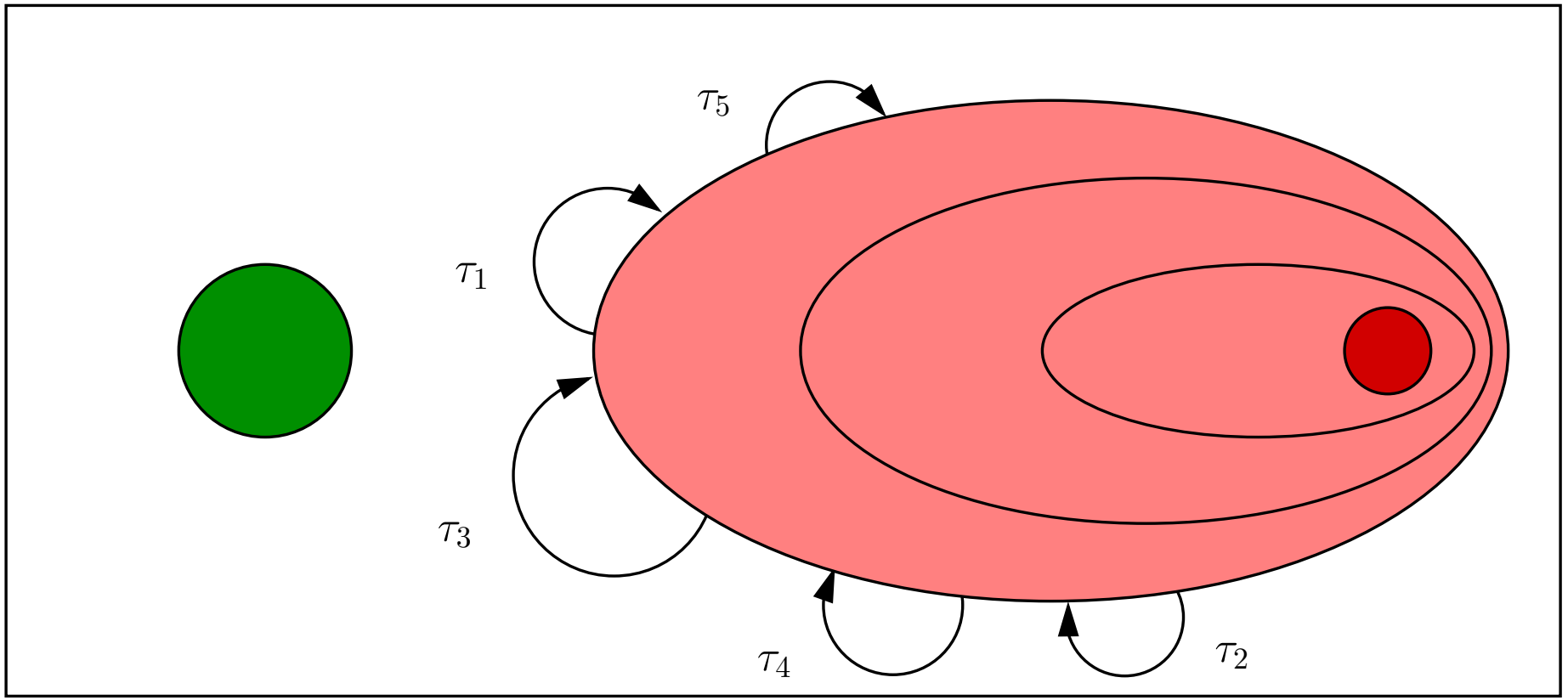
Initial States

Unsafe/Bad States

Reachable States

Reachable Bad States

System Status: **SAFE**



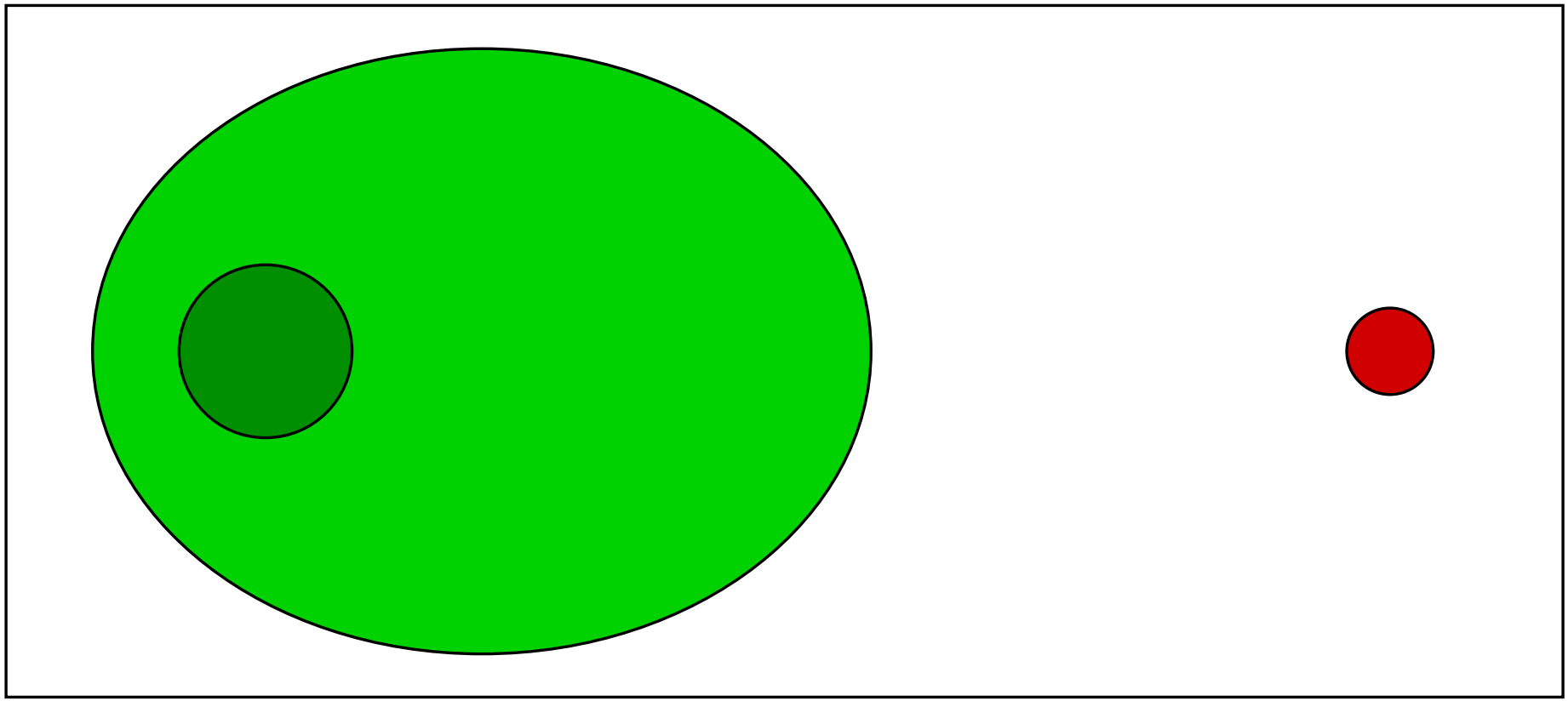
Initial States

Unsafe/Bad States

Reachable States

Reachable Bad States

System Status: `SAFE`



Initial States

Unsafe/Bad States

Reachable States

Reachable Bad States

Lamport's Mutual Exclusion Algorithm

The property MEX is verified almost istantaneously by MCMT[2]

Protocol	Property	Result	Time (s)
Lamport	MEX	`SAFE`	0.04

[2]Tools, encoded protocols, and results are available at
`http://www.oprover.org/mcmt_lynch_shavit.html`

Mutual Exclusion and Deadlock-Freedom

Consider the case in which we want to guarantee Deadlock-Freedom in addition to Mutual Exclusion.

There is a limitation of **asynchronous** algorithms (such as the Lamport's) with respect to DF which makes them impractical:

Theorem (Burns and Lynch '89)
There is no **asynchronous** algorithm providing MEX and DF for $n \geq 2$ processes using fewer than n shared read/write registers.

Solution: introduce the notion of **time** in the protocol (**timing-based** algorithms).

Timing-based Mutual Exclusion Algorithms

In a timing-based algorithm, each process has a **local notion of time**

In particular

- **TC1**: each process must execute a step in a given interval $[1, C]$
- **TC2**: a process can pause its execution with a $pause(delay)$ instruction, whose duration is in $(F, G]$

$$C, F, G$$

are 3 parameters (they will have no fixed value in our verification) subjected to the condition

$$1 \leq C \leq F \leq G$$

TC1 and **TC2** are called **Timing Constraints**

Timing-based Mutual Exclusion Algorithms

Algorithm 1

x, y: shared registers initially $y = 0$

repeat forever
0: *remainder exit$_i$*
1: $x := i$;
2: **if** $y \neq 0$ **then goto** 1;
3: $y := 1$;
4: **if** $x \neq i$ **then goto** 1;
5: *critical entry$_i$*
6: *critical exit$_i$*
7: $y := 0$;
8: *remainder entry$_i$*
end repeat

Lamport's

Algorithm 2

x: shared register, initially 0
delay: positive integer constant

repeat forever
0: *remainder exit$_i$*
1: **if** $x \neq 0$ **then goto** 1;
2: $x := i$;
3: *pause(delay)*;
4: **if** $x \neq i$ **then goto** 1;
5: *critical entry$_i$*
6: *critical exit$_i$*
7: $x := 0$;
8: *remainder entry$_i$*
end repeat

Fischer's

Algorithm 3

x, y: shared registers initially 0
delay: positive integer constant

repeat forever
0: *remainder exit$_i$*
1: **if** $x \neq 0$ **then goto** 1;
2: $x := i$;
3: *pause(delay)*;
4: **if** $x \neq i$ **then goto** 1;
5: **if** $y \neq 0$ **then goto** 1;
6: $y := 1$;
7: **if** $x \neq i$ **then goto** 1;
8: *critical entry$_i$*
9: *critical exit$_i$*
10: $y := 0$;
11: $x := 0$;
12: *remainder entry$_i$*
end repeat

Lynch-Shavit's[3]

Algorithm	TC respected	TC not respected
Fischer's	MEX, DF	DF
Lynch-Shavit	MEX, DF	MEX

[3] Algorithms 2, 3 first proposed in in *Timing-based mutual exclusion*, Lynch and Shavit.

Encoding Timing-based Algorithms

In order to handle the notion of **time** we extend array-based-systems into that of **parametrized timed systems**.

We introduce:

- an array pc_{clock}: $pc_{clock}[i]$ measures the time spent by process i between two program locations. It is reset to 0 when i changes location.
- suitable time constraints (e.g., $pc_{clock}[i] \leq C$) are added to the "standard" transitions

Encoding Timing-based Algorithms

In order to handle the notion of **time** we extend array-based-systems into that of **parametrized timed systems**.

We introduce:

- an array pc_{clock}: $pc_{clock}[i]$ measures the time spent by process i between two program locations. It is reset to 0 when i changes location.
- suitable time constraints (e.g., $pc_{clock}[i] \leq C$) are added to the "standard" transitions
- **time elapsing transitions** of the form

$$\exists \varepsilon \geq 0.\ (\forall j.\ \varphi(j, \underline{a}[j], pc_{clock}[j], \varepsilon)\ \wedge\ \underline{a}' = \underline{a} \wedge\ pc'_{clock} = \lambda j.\ (pc_{clock}[j] + \varepsilon))$$

which reads as

there is a positive time-elapse	$\exists \varepsilon \geq 0$
such that it holds the invariant	$\forall j.\ \varphi(j, \underline{a}[j], pc_{clock}[j], \varepsilon)$
the process stays the same	$\underline{a}' = \underline{a}$
but some time has passed	$pc'_{clock} = \lambda j.\ (pc_{clock}[j] + \varepsilon)$

Encoding Timing-based Algorithms

In order to handle the notion of **time** we extend array-based-systems into that of **parametrized timed systems**.

We introduce:

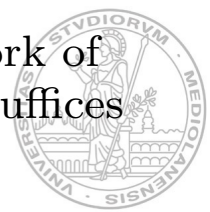
- an array pc_{clock}: $pc_{clock}[i]$ measures the time spent by process i between two program locations. It is reset to 0 when i changes location.
- suitable time constraints (e.g., $pc_{clock}[i] \leq C$) are added to the "standard" transitions
- **time elapsing transitions** of the form

$$\exists \varepsilon \geq 0.\ (\forall j.\ \varphi(j, \underline{a}[j], pc_{clock}[j], \varepsilon)\ \wedge\ \underline{a}' = \underline{a} \wedge\ pc'_{clock} = \lambda j.\ (pc_{clock}[j] + \varepsilon))$$

  which reads as

there is a positive time-elapse	$\exists \varepsilon \geq 0$
such that it holds the invariant	$\forall j.\ \varphi(j, \underline{a}[j], pc_{clock}[j], \varepsilon)$
the process stays the same	$\underline{a}' = \underline{a}$
but some time has passed	$pc'_{clock} = \lambda j.\ (pc_{clock}[j] + \varepsilon)$

- Remark: the guard $\forall j.\ \varphi(j, \underline{a}[j], pc_{clock}[j], \varepsilon)$ is "problematic" for the framework of MCMT: an approximated analysis is used (stopping failure model) which still suffices for proving properties



MEX Verification for Fischer and Lynch-Shavit

Algorithm	TC respected	TC not respected
Fischer's	MEX, DF	DF
Lynch-Shavit	MEX, DF	MEX

Protocol	Property	Result	Time (s)	Notes
Fischer	MEX	`SAFE`	2.64	TC specified
	MEX	`UNSAFE`	3.73	TC not specified
	MEX + I1	`SAFE`	(0.02 + 0.17) 0.19	Invariant added
Lynch-Shavit	MEX	`SAFE`	24.39	TC specified
	MEX	`SAFE`	353.91	TC not specified
	MEX abstr.	`SAFE`	8.56	Uses MCMT's abstraction

Deadlock-Freedom via Verification and Synthesis of Waiting-time Constraints

Deadlock-Freedom and Waiting time constraints

Notice that Deadlock-Freedom **is not a safety property**. However it can be shown that:[4]

Observation
It is possible to derive an upper bound for an arbitrary process to enter the critical region. This upper bound is independent from the number running processes, and it is a linear polynome $p(C, G)$ in the parameters C and G.

[4] See e.g. Chapter 24 of book *Distributed Algorithms*, Nancy Lynch.

Deadlock-Freedom and Waiting time constraints

Notice that Deadlock-Freedom **is not a safety property**. However it can be shown that:[4]

Observation
It is possible to derive an upper bound for an arbitrary process to enter the critical region. This upper bound is independent from the number running processes, and it is a linear polynome $p(C, G)$ in the parameters C and G.

Optimal polynomes were (manually) shown to be

- $p(C, G) = 2G + 5C$ for Fischer's
- $p(C, G) = 2G + 9C$ for Lynch-Shavit's

[4] See e.g. Chapter 24 of book *Distributed Algorithms*, Nancy Lynch.

Deadlock-Freedom and Waiting time constraints

Notice that Deadlock-Freedom **is not a safety property**. However it can be shown that:[4]

Observation
It is possible to derive an upper bound for an arbitrary process to enter the critical region. This upper bound is independent from the number running processes, and it is a linear polynome $p(C, G)$ in the parameters C and G.

Optimal polynomes were (manually) shown to be

- $p(C, G) = 2G + 5C$ for Fischer's
- $p(C, G) = 2G + 9C$ for Lynch-Shavit's

Deadlock-Freedom can be recast as

DF: **from any reachable state** such that

(i) some process is in the *trying region*

(ii) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

[4] See e.g. Chapter 24 of book *Distributed Algorithms*, Nancy Lynch.

Verification of Waiting time constraints

DF: **from any reachable state** such that

(*i*) some process is in the *trying region*

(*ii*) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

We automatically verify the optimal bounds by running the following **auxiliary verification problem** which has the same transitions of the protocol, but:

[5] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Verification of Waiting time constraints

DF: **from any reachable state** such that

(i) some process is in the *trying region*

(ii) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

We automatically verify the optimal bounds by running the following **auxiliary verification problem** which has the same transitions of the protocol, but:

I. We introduce a shared absolute clock abs_{clock} and a Boolean flag k ($k = true$ means some process has entered the critical section)

[5] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Verification of Waiting time constraints

DF: **from any reachable state** such that

(*i*) some process is in the *trying region*

(*ii*) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

We automatically verify the optimal bounds by running the following **auxiliary verification problem** which has the same transitions of the protocol, but:

I. We introduce a shared absolute clock abs_{clock} and a Boolean flag k ($k = true$ means some process has entered the critical section)

II. We set the initial states as

(*a*) $abs_{clock} = 0$ and $k = false$

(*b*) process 1 is in *trying* and no process is in *critical*

[5] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Verification of Waiting time constraints

DF: **from any reachable state** such that

(*i*) some process is in the *trying region*

(*ii*) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

We automatically verify the optimal bounds by running the following **auxiliary verification problem** which has the same transitions of the protocol, but:

I. We introduce a shared absolute clock abs_{clock} and a Boolean flag k ($k = true$ means some process has entered the critical section)

II. We set the initial states as
   (*a*) $abs_{clock} = 0$ and $k = false$
   (*b*) process 1 is in *trying* and no process is in *critical*

III. We declare the unsafe states as $abs_{clock} > p(C, G)$ and $k = false$

[5] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Verification of Waiting time constraints

DF: **from any reachable state** such that

(*i*) some process is in the *trying region*

(*ii*) no process is in the *critical region*

it is never the case that $p(C, G)$ time passes before a process enters *the critical region*.

We automatically verify the optimal bounds by running the following **auxiliary verification problem** which has the same transitions of the protocol, but:

I. We introduce a shared absolute clock abs_{clock} and a Boolean flag k ($k = true$ means some process has entered the critical section)

II. We set the initial states as
   (*a*) $abs_{clock} = 0$ and $k = false$
   (*b*) process 1 is in *trying* and no process is in *critical*

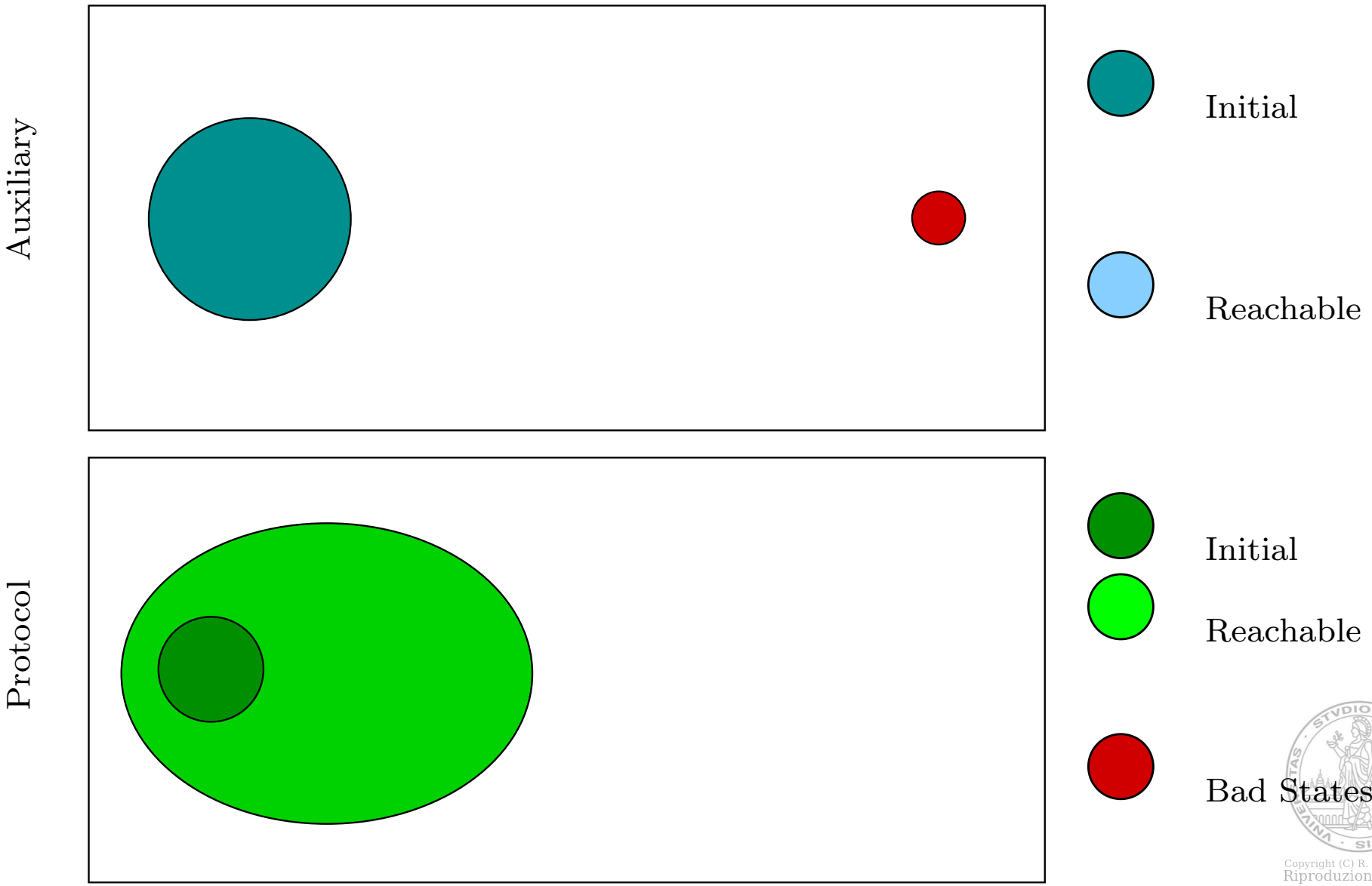
III. We declare the unsafe states as $abs_{clock} > p(C, G)$ and $k = false$

IV. **We add "manually derived" invariants to overapproximate the set of reachable states**[5]

[5] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

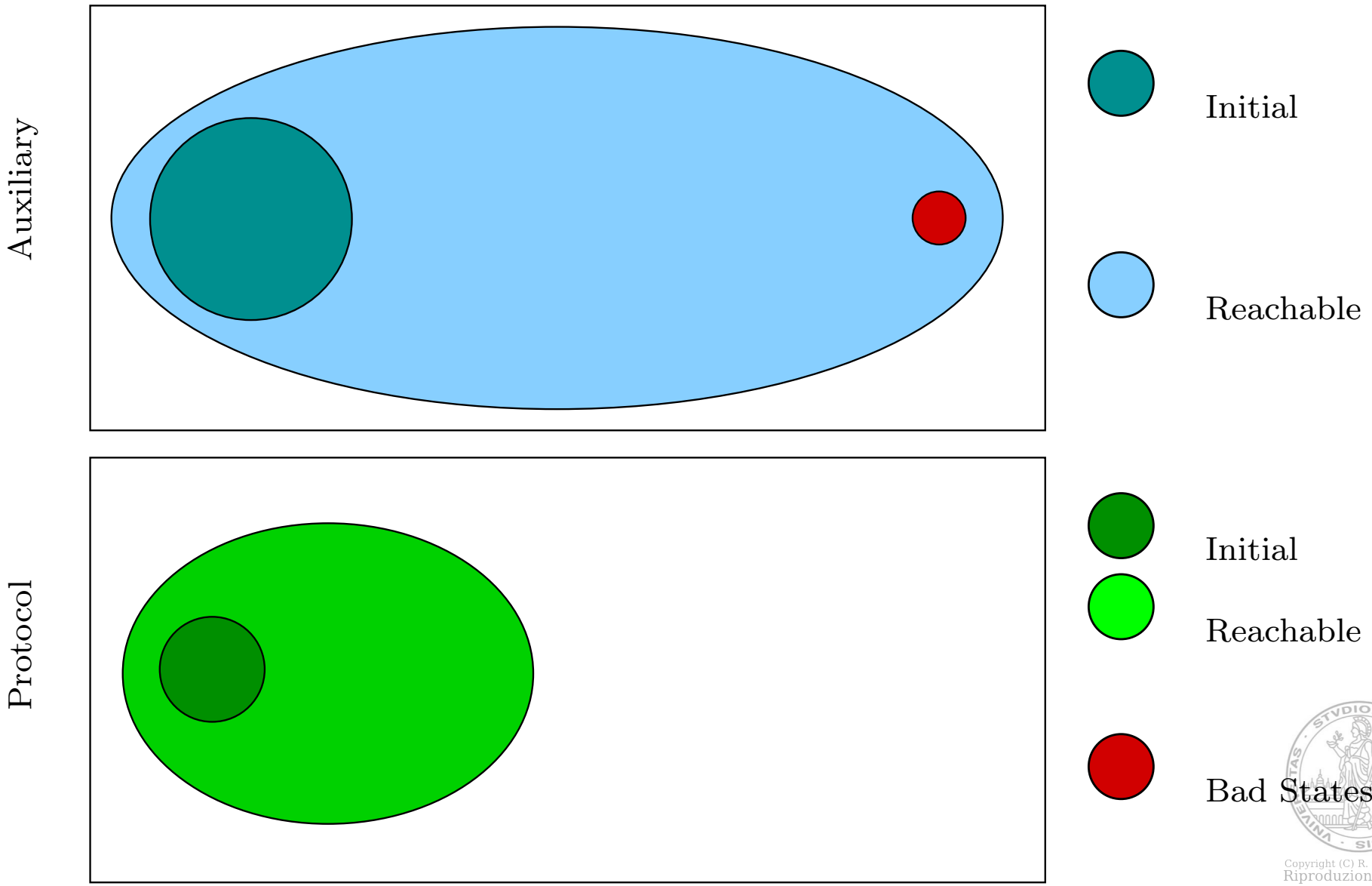
Verification of Waiting time constraints

System Status: `UNKNOWN`



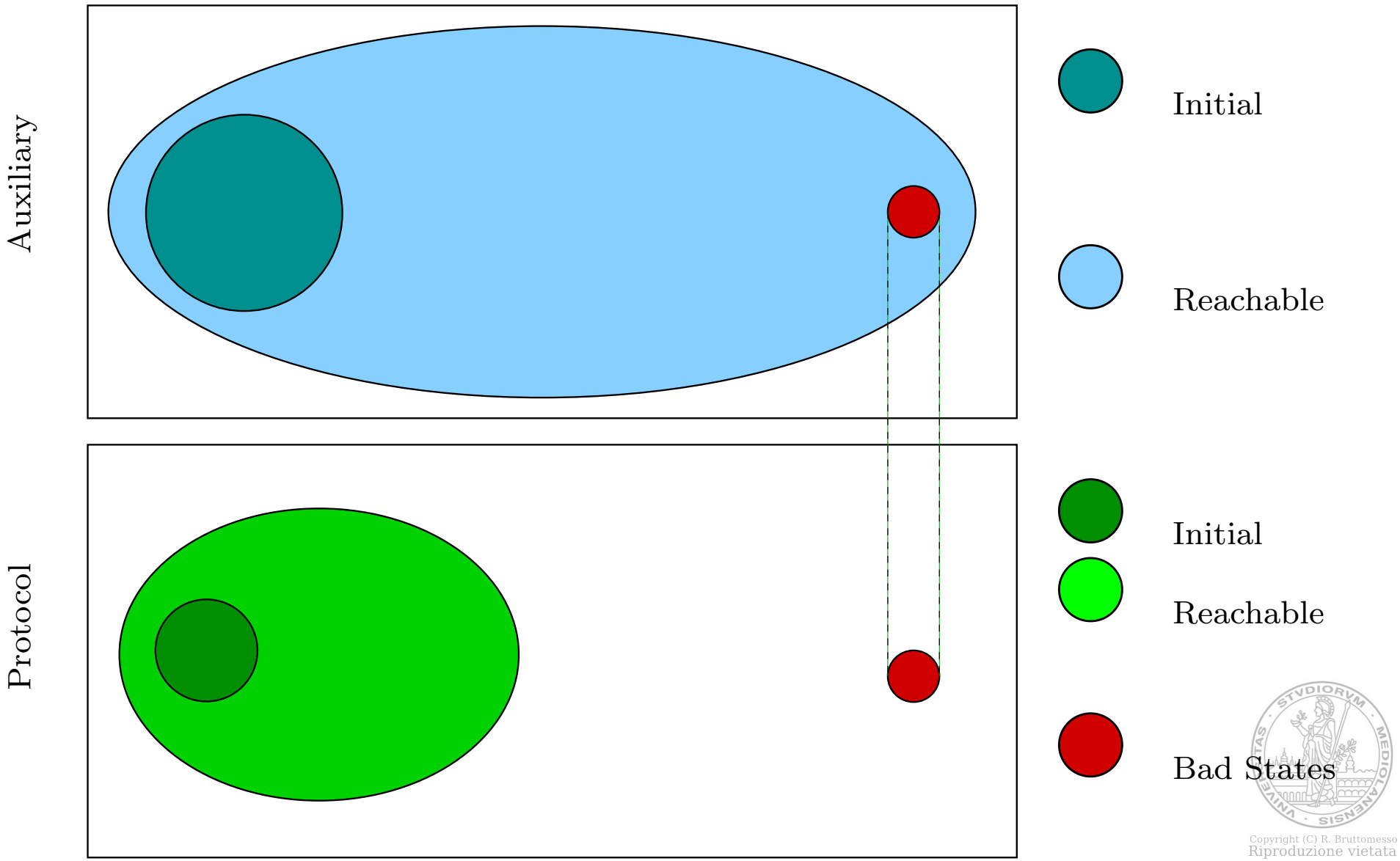
Verification of Waiting time constraints

System Status: `UNSAFE ?`



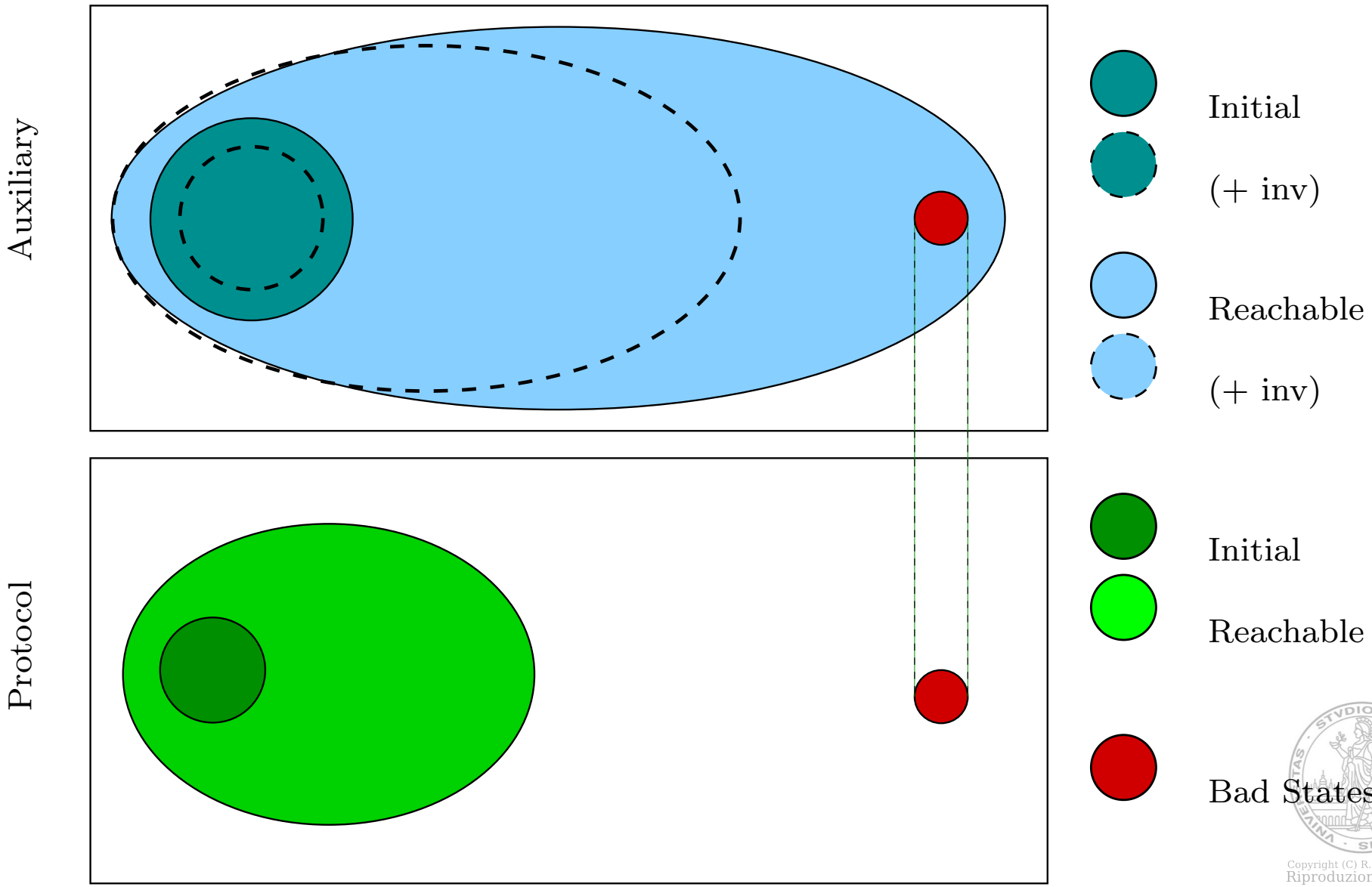
Verification of Waiting time constraints

System Status: `UNSAFE` ? No, false alarm !



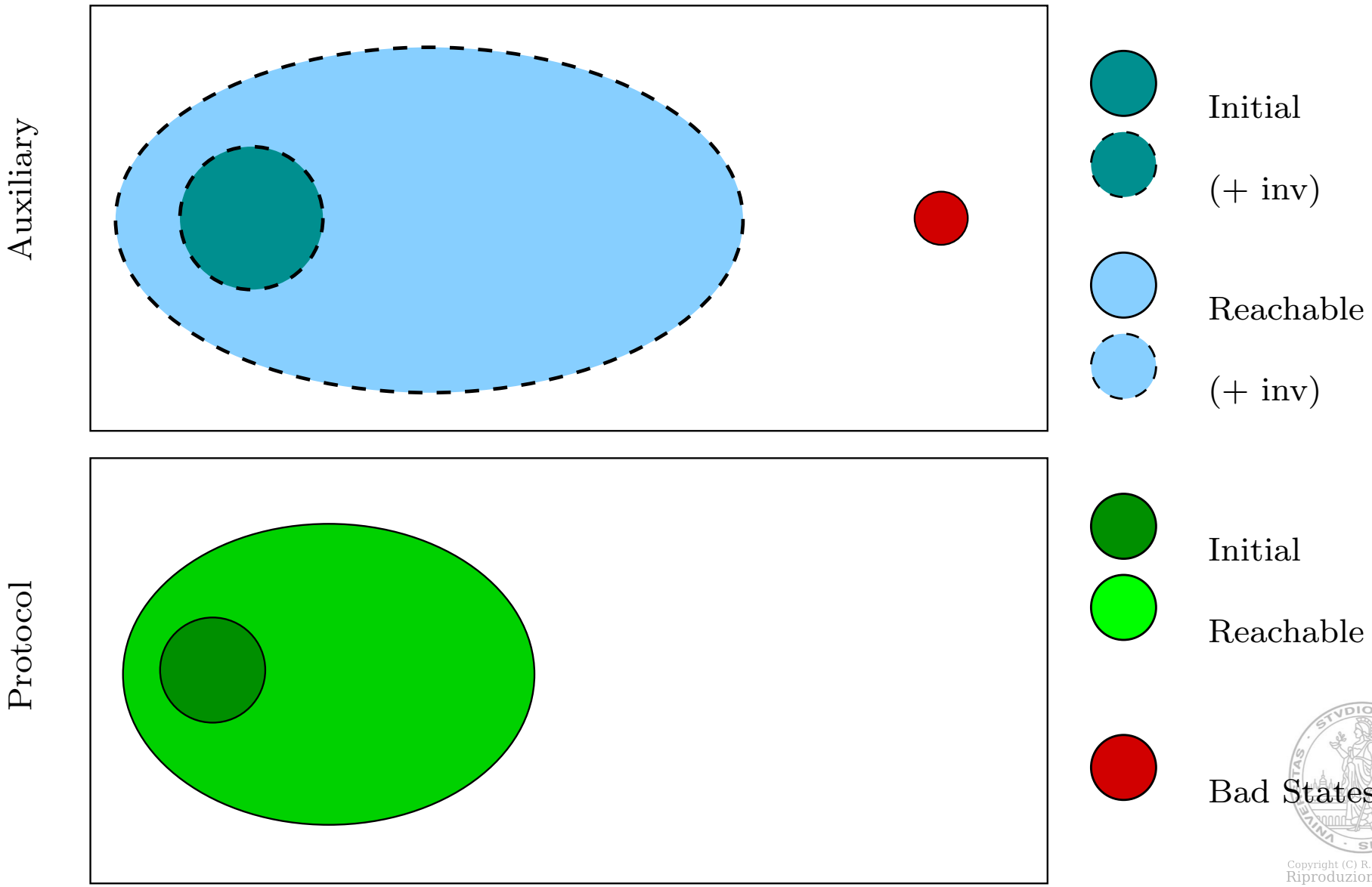
Verification of Waiting time constraints

System Status:



Verification of Waiting time constraints

System Status: **SAFE**



Automatic Verification of Waiting time constraints

(*IV*) **We add "manually derived" invariants to overapproximate the set of reachable states**[6]

[6] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Automatic Verification of Waiting time constraints

(*IV*) **We add "manually derived" invariants to overapproximate the set of reachable states**[6]

It is also possible to mechanize the generation of the "manually derived" invariants. The stragegy consists in running two different safety problems, say a **guesser** and a **checker**.

guesser (the Auxiliary System):

- Search for states that violates the provided polynome $p(C, G)$
- If "violating states" are found, pass them to **checker**
- Otherwise $p(C, G)$ holds !

[6] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

Automatic Verification of Waiting time constraints

(*IV*) **We add "manually derived" invariants to overapproximate the set of reachable states**[6]

It is also possible to mechanize the generation of the "manually derived" invariants. The stragegy consists in running two different safety problems, say a **guesser** and a **checker**.

guesser (the Auxiliary System):

- Search for states that violates the provided polynome $p(C, G)$
- If "violating states" are found, pass them to **checker**
- Otherwise $p(C, G)$ holds !

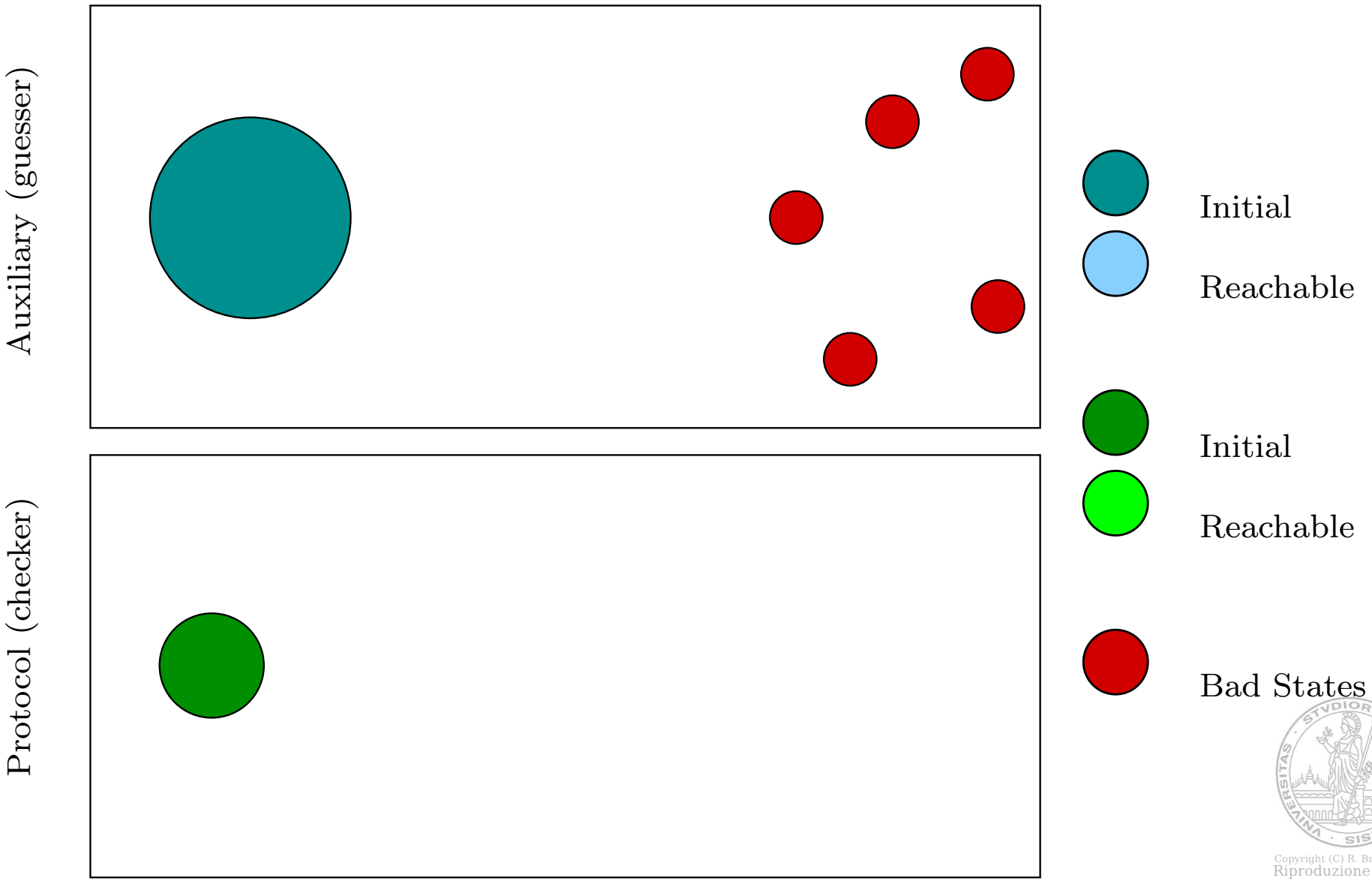
checker (the Original Protocol):

- checks if "violating states" provided by guesser are actually reachable in the original system
- If they are reachable, than $p(C, G)$ does not hold !
- Otherwise the negation of "violating states" are actually an invariant for the protocol
- Add this invariant to the **guesser** and run it again

[6] These invariants are used in proofs in *Timing-based mutual exclusion*, Lynch and Shavit.

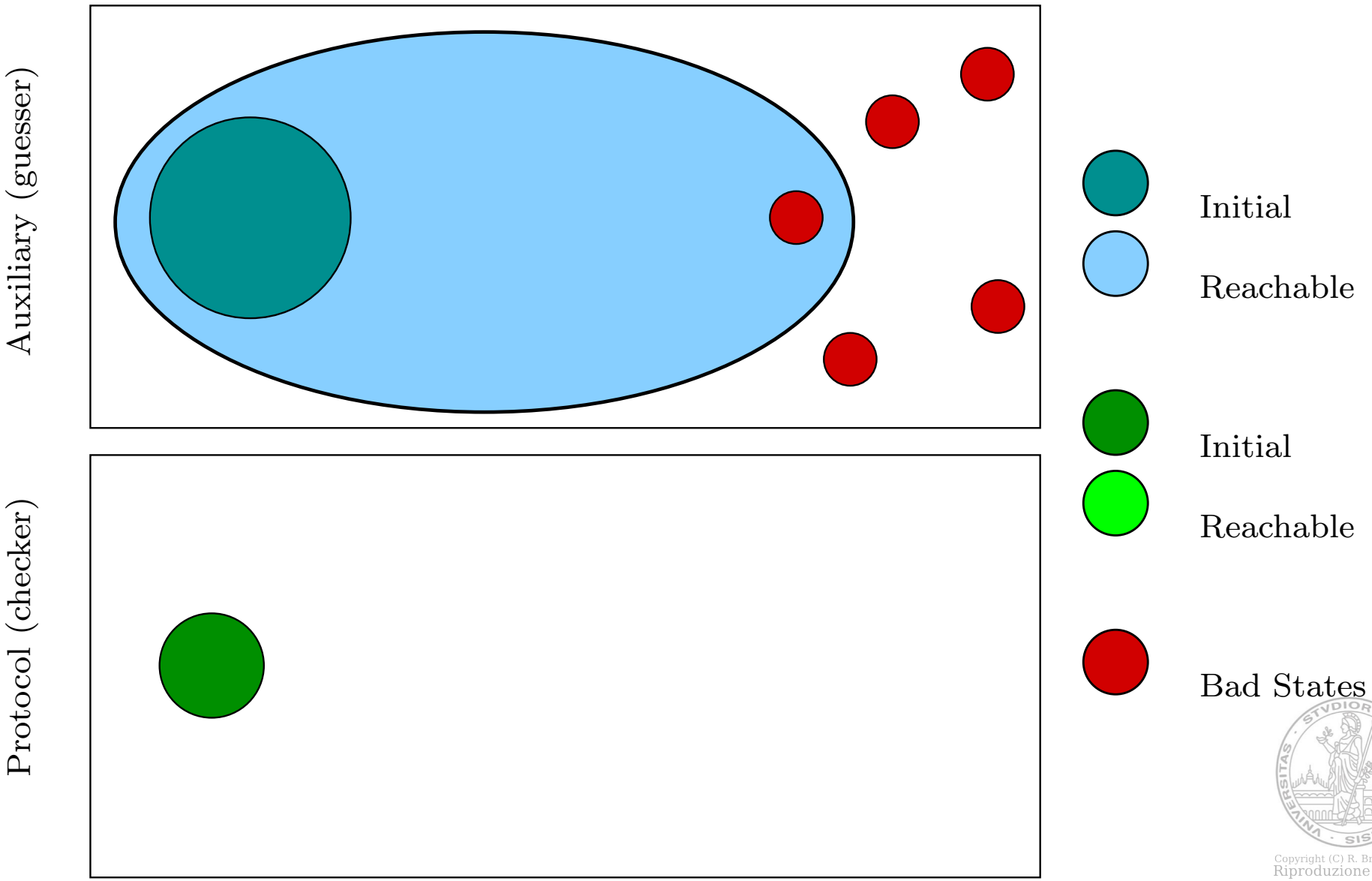
Automatic Verification of Waiting time constraints

System Status: `UNKNOWN`



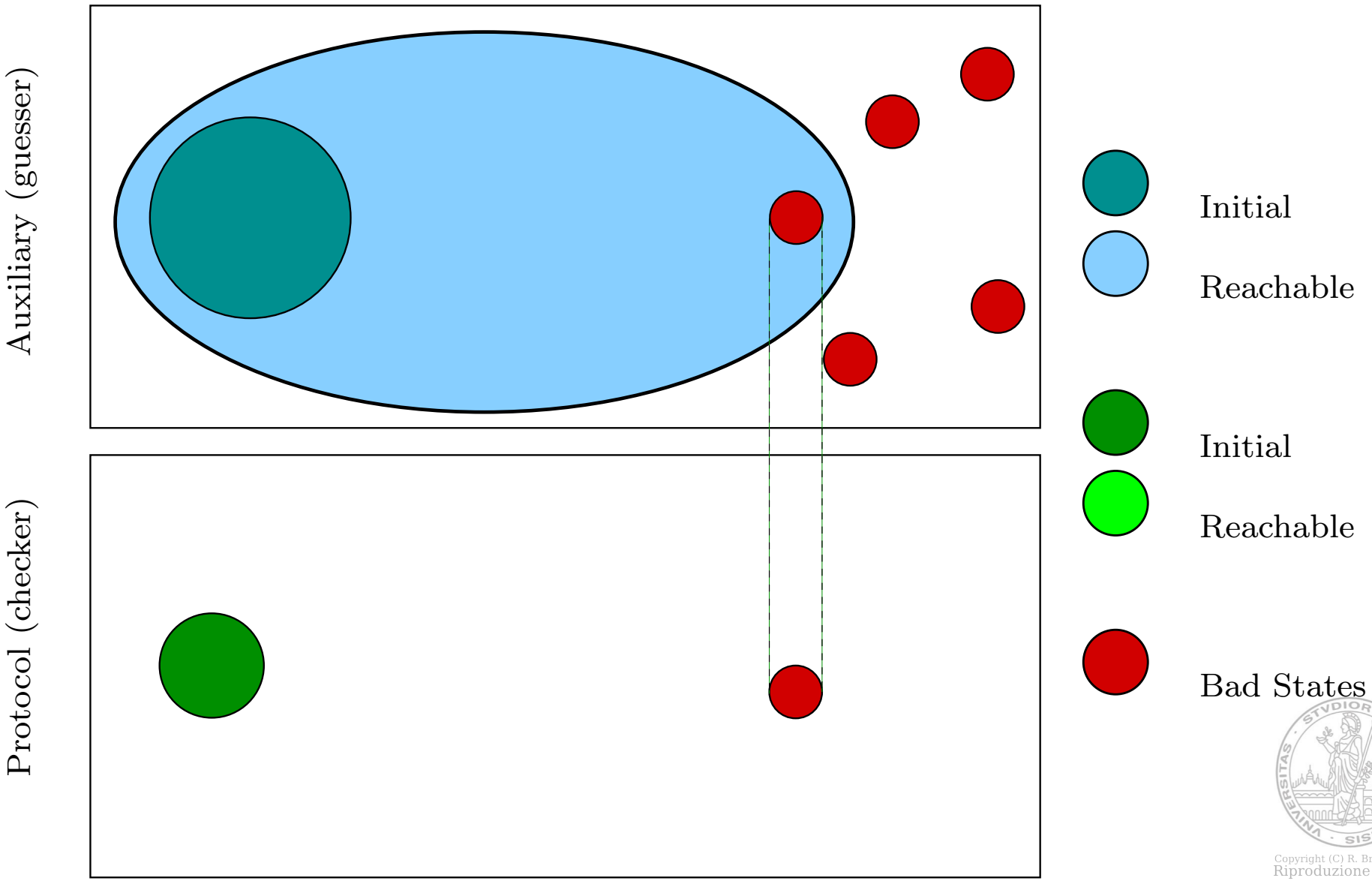
Automatic Verification of Waiting time constraints

System Status: `UNSAFE ?`



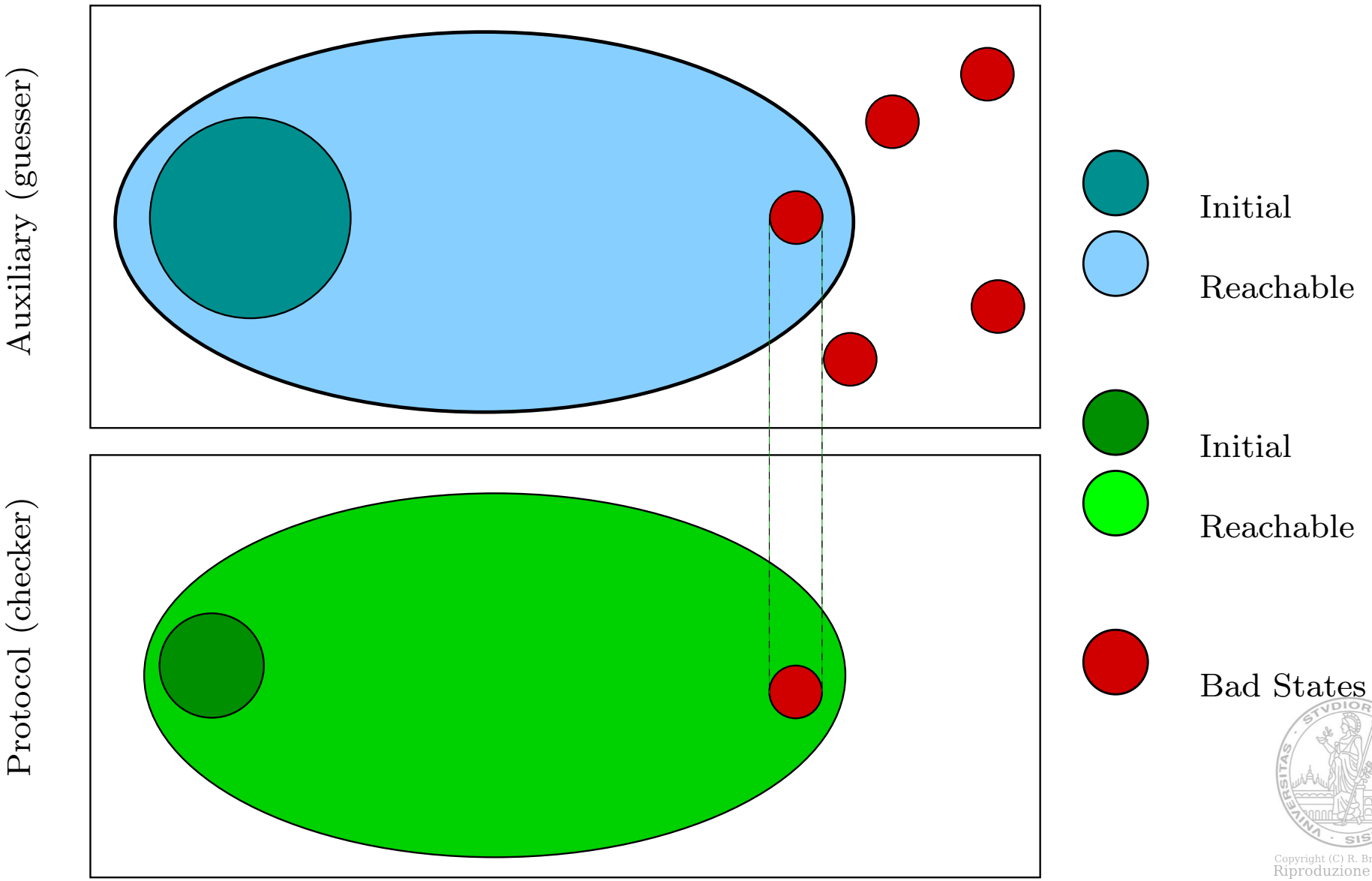
Automatic Verification of Waiting time constraints

System Status: `UNSAFE ?`



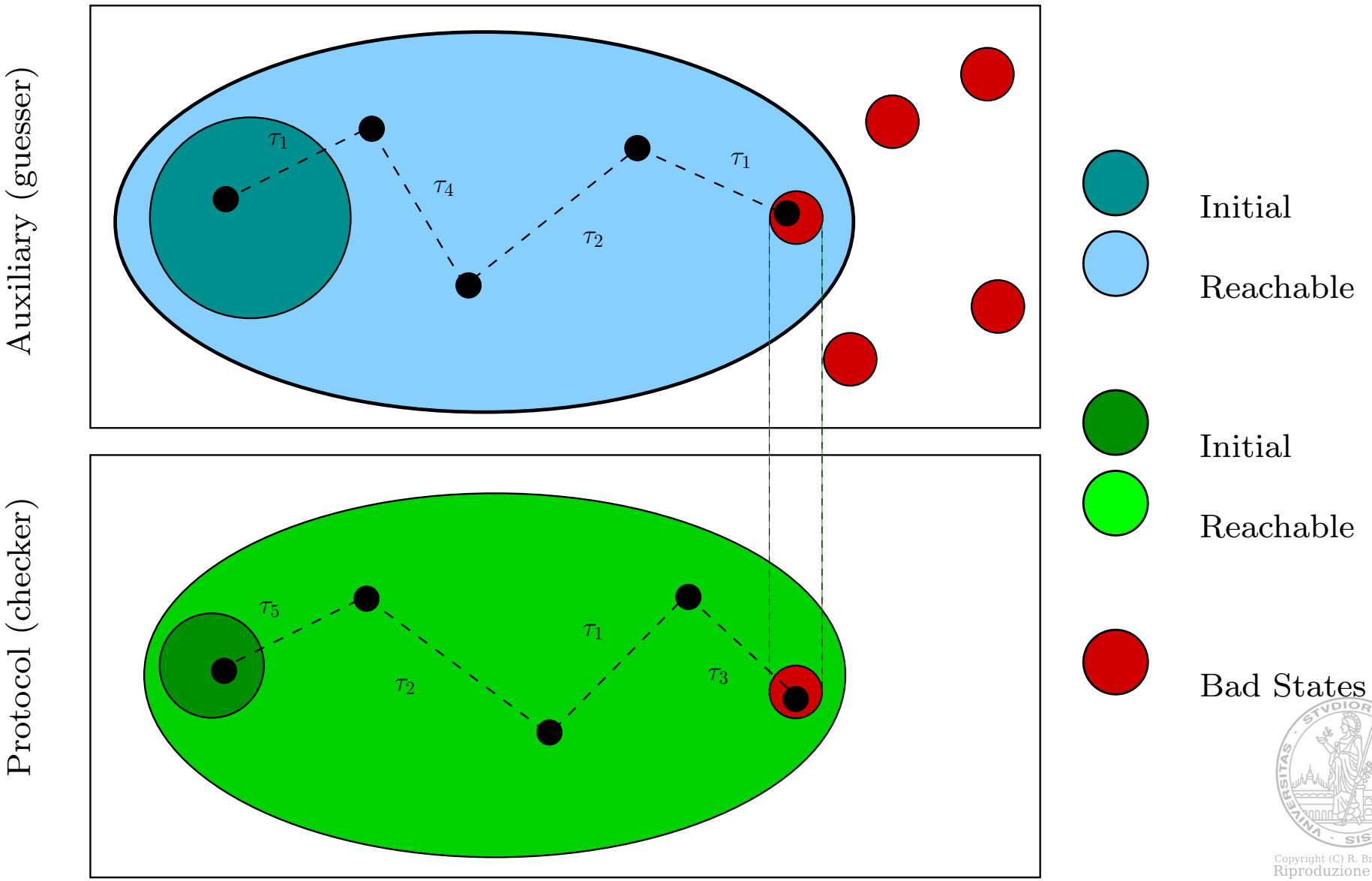
Automatic Verification of Waiting time constraints

System Status: `UNSAFE` ? Yes, it is really so



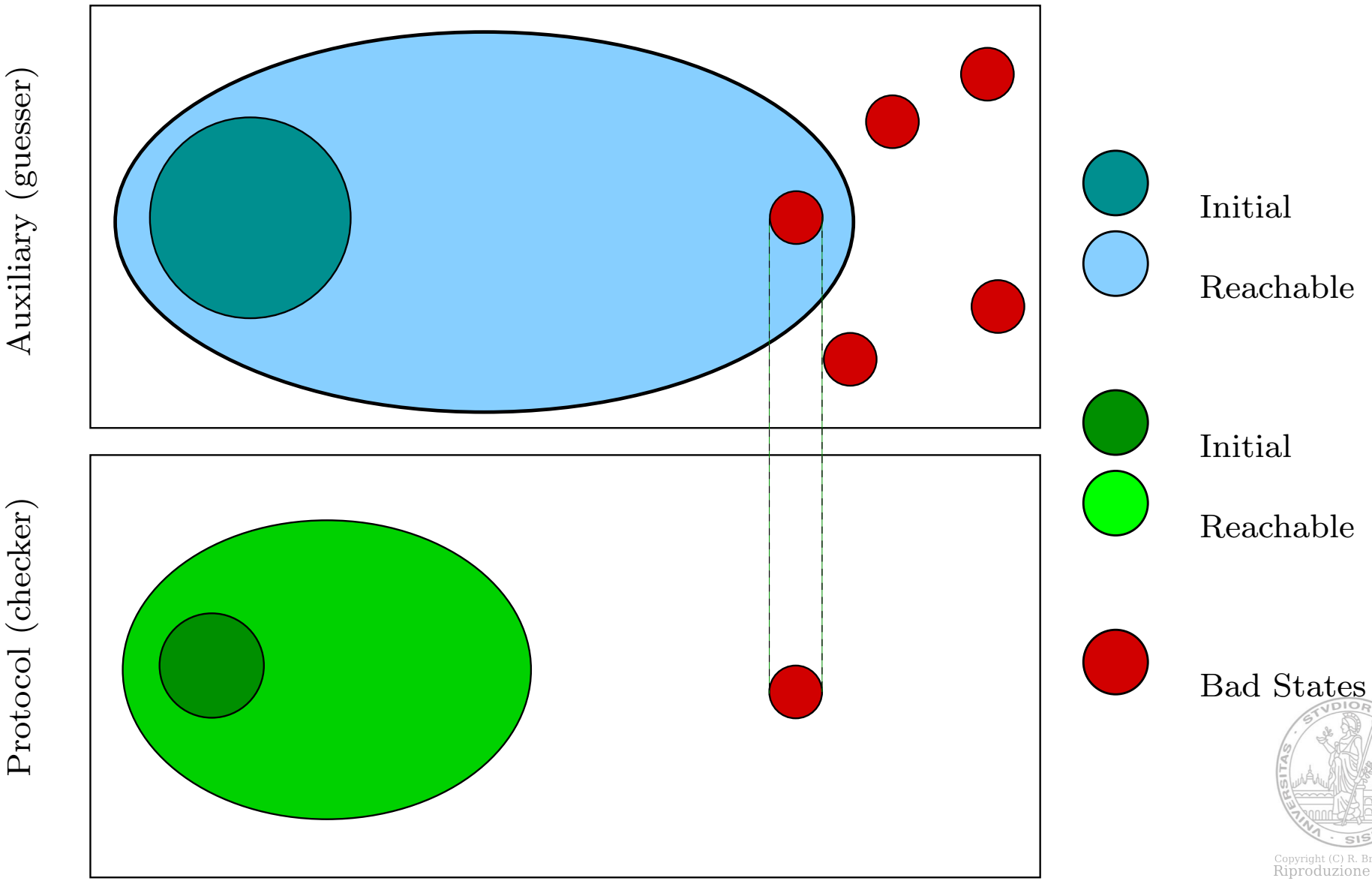
Automatic Verification of Waiting time constraints

System Status: `UNSAFE` ? Yes, it is really so



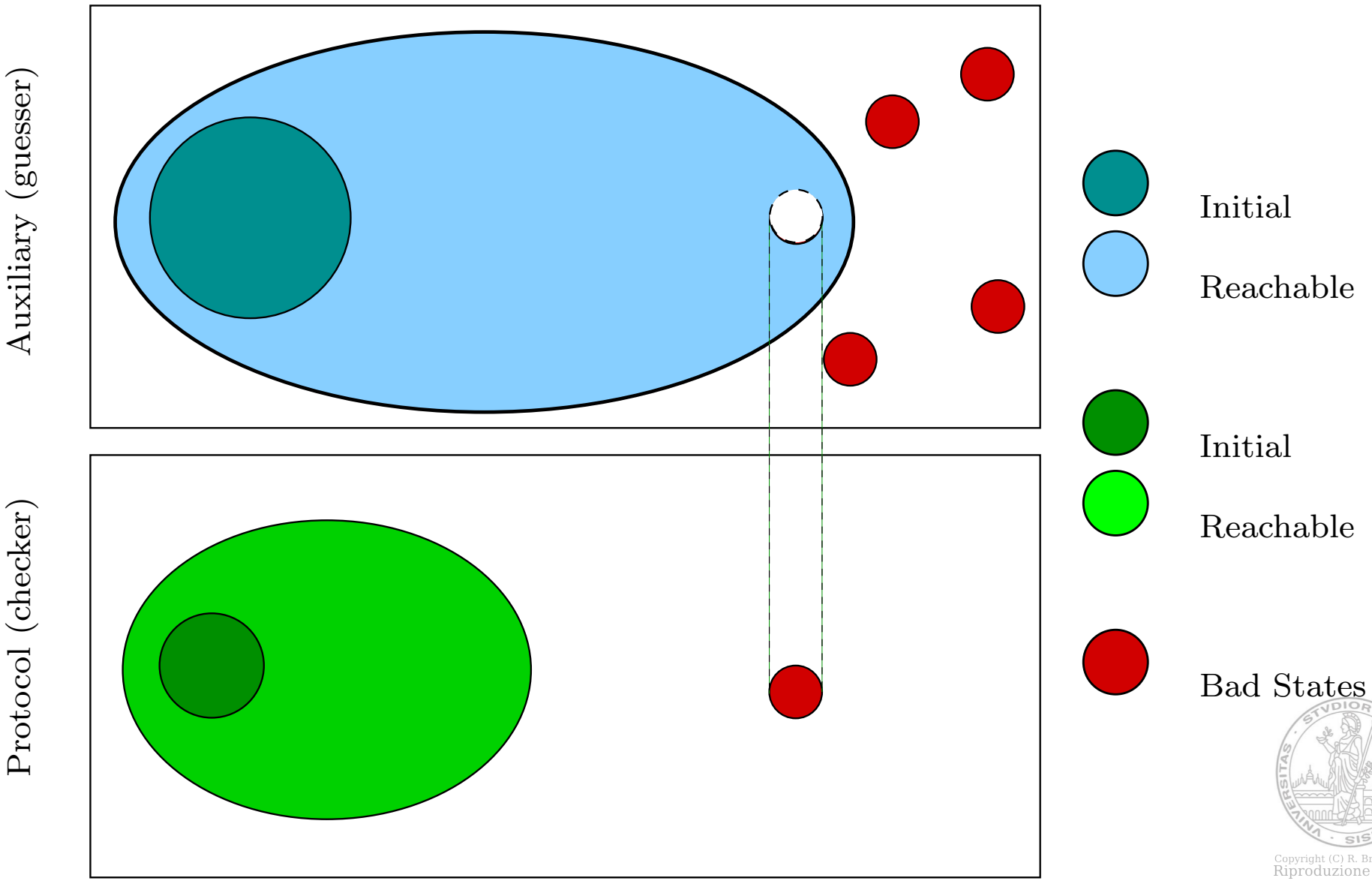
Automatic Verification of Waiting time constraints

System Status: `UNSAFE` ? No, it is a false alarm



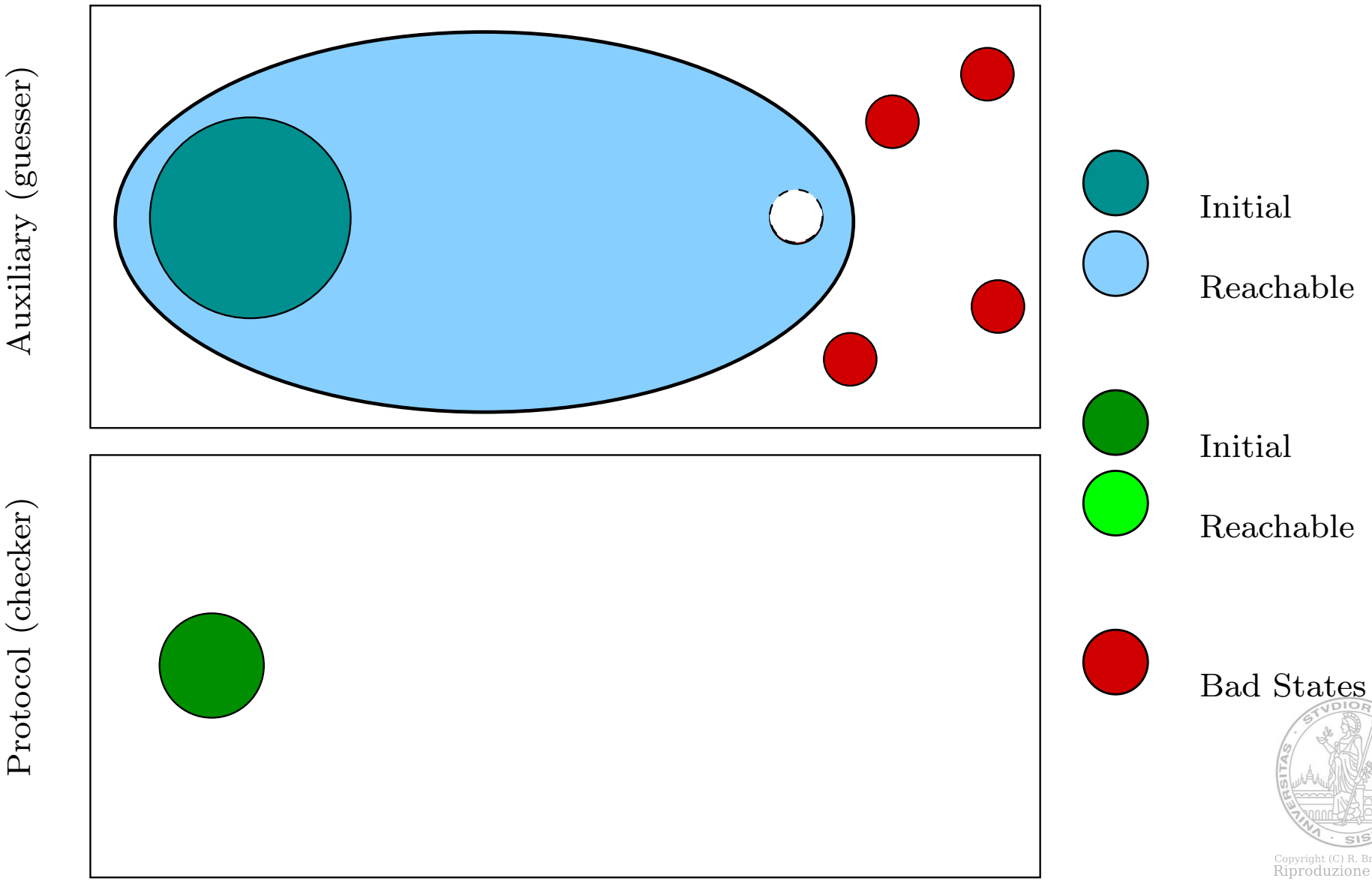
Automatic Verification of Waiting time constraints

System Status: **Adding Negation of Bad State as Invariant !**



Automatic Verification of Waiting time constraints

System Status: `UNKNOWN`



Automatic Synthesis of Waiting time constraints

Suppose that the polynomes were unknown to us. Using the previous "Guess-Check" procedure as sub-routine we can synthesize them given a template

$$p(C, G) = \alpha G + \beta C$$

by running a sort of "binary search" on the values α, β

Automatic Synthesis of Waiting time constraints

Suppose that the polynomes were unknown to us. Using the previous "Guess-Check" procedure as sub-routine we can synthesize them given a template

$$p(C, G) = \alpha G + \beta C$$

by running a sort of "binary search" on the values α, β

α, β	Bound Holds	Guess-Check Iters	Time (s)
Fischer			
2, 2	NO	1	62.06
2, 4	NO	5	110.68
2, 5	YES	8	155.56
2, 6	YES	6	130.69
2, 10	YES	3	51.25
Lynch-Shavit			
2, 2	NO	1	224.74
2, 8	NO	11	5764.42
2, 9	YES	16	27995.78
2, 10	YES	10	6935.91
2, 14	YES	3	974.06

Conclusion

Conclusion

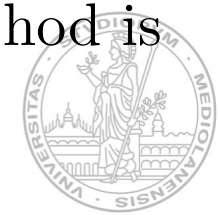
We have shown a set of techniques to automatically verify parametrized timed systems

We have applied them to the non-trivial verification of the Fischer and Lynch-Shavit protocols for an unbounded number of running processes

but they may apply to other timed cases as well (they are generic, not specific to the two test cases presented)

These techniques include automatic verification of Deadlock-Freedom by reduction to a number of sub-calls to a safety verification problem

Experiments were run with the tool MCMT showing that the method is viable and efficient in practice



Thanks for your attention

Fischer's Mutual Exclusion Algorithm

- Transition τ_3

$$
\begin{array}{ll}
\exists i.\ (& pc[i] = 1 \wedge \\
& pc_{clock}[i] \geq 1 \wedge \\
& x \neq 0 \wedge \\
& \\
& x' = x \wedge \\
& pc' = \lambda j.\ \mathrm{ite}(i = j,\ 1,\ pc[j]) \wedge \\
& pc'_{clock} = \lambda j.\ \mathrm{ite}(i = j,\ 0,\ pc_{clock}[j]) \\
) &
\end{array}
$$

Algorithm 2

x: shared register, initially 0
delay: positive integer constant

repeat forever
0: *remainder* $exit_i$
1: **if** $x \neq 0$ **then goto** 1;
2: $x := i$;
3: *pause(delay)*;
4: **if** $x \neq i$ **then goto** 1;
5: *critical* $entry_i$
6: *critical* $exit_i$
7: $x := 0$;
8: *remainder* $entry_i$
end repeat

Verification of Waiting time constraints

Protocol	Property	Result	Time (s)	
Fischer	DF + Inv.	`SAFE`	(8.95 + 80.97)	89.92
Lynch-Shavit	DF + Inv.	`SAFE`	(236.51 + 1374.38)	1610.89