CS:4980
Foundations of Embedded Systems

Liveness Requirements

Part II

LTL Recap

Syntax: Formulas built from

- Base formulas: Boolean-valued expressions over typed variables
- Logical connectives: $\wedge$, $\vee$, $\Rightarrow$, $\neg$, …
- Temporal Operators: Always, Eventually, Next, Until

Semantics: defined by rules for the satisfaction relation

- Formulas are evaluated w.r.t. a trace ρ (infinite sequence of valuations)
- A system satisfies spec φ iff every infinite execution satisfies φ

Derived operators: Repeatedly (Always Eventually)
Persistently (Eventually Always)

Sample requirement: Every request is eventually granted

Always [req = 1 $\Rightarrow$ Eventually (grant = 1)]

Temporal Implications and Equivalences

Understanding subtle differences among different variants of LTL formulas can be tricky

Definition: Let φ, ψ be LTL formulas

1. φ is *stronger* than ψ if every trace that satisfies φ satisfies ψ too
   - i.e., every trace satisfies the implication $\varphi \Rightarrow \psi$
2. φ is equivalent to ψ if φ and ψ are satisfied by exactly the same traces
   - i.e., each formula is stronger than the other
   - i.e., every trace satisfies the double implication $\varphi \Leftrightarrow \psi$
   - i.e., the two formulas express exactly the same requirement

Knowing some standard equivalences is useful for simplifying formulas

Temporal Implications and Equivalences

- Always φ is stronger than φ
- Repeatedly φ is equivalent to ¬Persistently ¬φ
- Persistently φ is stronger than Repeatedly φ
- Always φ is equivalent to φ ∧ Next Always φ
- Always φ is equivalent to ¬Eventually ¬φ

Exercise: What is the mutual relationship between these formulas?

- Always Eventually φ
- Next Always Eventually φ
- Eventually Always Eventually φ

Logical Connectives and Temporal Operators

Are these two formulas equivalent?

Eventually $(\varphi \vee \psi)$ and Eventually $\varphi \vee$ Eventually ψ

Yes, they are.

Proof:

$\Rightarrow$) Suppose a trace ρ satisfies Eventually $(\varphi \vee \psi)$

- There is a position j such that $(\rho, j) \vDash \varphi \vee \psi$
- Either $(\rho, j) \vDash \varphi$ or $(\rho, j) \vDash \psi$
- Suppose $(\rho, j) \vDash \varphi$ (the other case is similar)
- Then ρ satisfies Eventually φ
- Hence it also satisfies Eventually $\varphi \vee$ Eventually ψ

$\Leftarrow$) Suppose a trace ρ satisfies Eventually $\varphi \vee$ Eventually ψ

- Suppose ρ satisfies Eventually φ (the other case is similar)
- There is a position j such that $(\rho, j) \vDash \varphi$
- Then, $(\rho, j) \vDash \varphi \vee \psi$
- It follows that ρ satisfies Eventually $(\varphi \vee \psi)$

Logical Connectives and Temporal Operators

Are these two formulas equivalent?

Eventually $(\varphi \wedge \psi)$ and Eventually $\varphi \wedge$ Eventually ψ

The first is stronger than the second but not vice versa

Proof:

$\Rightarrow$) Suppose a trace ρ satisfies Eventually $(\varphi \wedge \psi)$

- There exists a position j such that $(\rho, j) \models \varphi \wedge \psi$
- It follows that both $(\rho, j) \models \varphi$ and $(\rho, j) \models \psi$
- Since $(\rho, j) \models \varphi$ then ρ satisfies Eventually φ
- Similarly, it also satisfies Eventually ψ
- It follows that ρ satisfies Eventually $\varphi \wedge$ Eventually ψ

$\Leftarrow$) To disprove this, consider trace 0,1,0,1,0,1,... over a Boolean variable x

- Trace satisfies Eventually $(x = 0) \wedge$ Eventually $(x = 1)$
- But does not satisfy Eventually $(x = 0 \wedge x = 1)$

Logical Connectives and Temporal Operators

Distributivity rules for logical connectives and temporal operators

Exercise: Are these equivalent?

- Always ($\varphi \wedge \psi$) and Always $\varphi \wedge$ Always ψ
- Always ($\varphi \vee \psi$) and Always $\varphi \vee$ Always ψ
- Repeatedly ($\varphi \wedge \psi$) and Repeatedly $\varphi \wedge$ Repeatedly ψ
- Repeatedly ($\varphi \vee \psi$) and Repeatedly $\varphi \vee$ Repeatedly ψ

Back to Fairness

Weak fairness: An infinite execution is *fair* to a task A if, repeatedly, either A is executed or is disabled

If task is enabled, then it is eventually executed or disabled

Strong fairness: An infinite execution is *fair* to a task A, if task A is either executed repeatedly or disabled continuously from a certain step onwards

If task is repeatedly enabled, then it is repeatedly executed

Back to Fairness

Process P

nat x := 0; bool y := 0
A: x := x + 1 B: even(x) –> y := 1 - y

What fairness assumptions are needed so that P satisfies the spec

- Eventually $(x \geq 10)$: weak fairness for A
- Eventually $(y = 1)$: strong fairness for B

Back to Fairness

Process P

```
nat x := 0;  bool y := 0
A:  x := x + 1
B:  even(x) –> y := 1 - y
```

- Fairness can be encoded directly in LTL!
- Instead of checking if the system satisfies an LTL formula φ, check if it satisfies the formula

$$\text{FairnessAssumption} \Rightarrow \varphi$$

- FairnessAssumption is an LTL formula encoding what it means for an execution to be weakly/strongly fair with respect to a task

Encoding Weak Fairness in LTL

Process P

```
nat x := 0;  bool y := 0;  {A,B} executed
A:  x := x + 1;   executed := A
B:  even(x) –> y := 1 - y;  executed := B
```

- We add a variable called executed whose values are task names
- Whenever a task executes, executed is assigned the name of the task

Weak fairness for a task T: An infinite execution is weakly fair to task T if it satisfies the formula

WF(T): Persistently (T is enabled) ⇒ Repeatedly (T is executed)

Examples:

WF(A): Repeatedly (executed = A)

WF(B): Persistently (even(x)) ⇒ Repeatedly (executed = B)

Checking Requirements under Weak Fairness

Process P

nat x := 0; bool y := 0; {A,B} executed
A: x := x + 1; executed := A B: even(x) –> y := 1 - y; executed := B

Does P satisfy

1. Eventually ($x \geq 10$) ?
2. WF(A) $\Rightarrow$ Eventually ($x \geq 10$) ?
3. WF(B) $\Rightarrow$ Eventually ($y = 1$) ?
4. (WF(A) $\wedge$ WF(B)) $\Rightarrow$ Eventually ($y = 1$) ?

What have we achieved?

- Checking if an LTL spec is satisfied under fairness assumptions is reduced to checking a modified LTL spec
- Then the verifier does not have to handle fairness explicitly

Encoding Strong Fairness

Process P

```
nat x := 0;  bool y := 0;  {A,B} executed
A:  x := x + 1;   executed := A
B:  even(x) –> y := 1 - y;  executed := B
```

Strong fairness for a task T: An infinite execution is strongly fair to task T if it satisfies the formula

SF(T): Repeatedly (T is enabled) ⇒ Repeatedly (T is executed)

Example:

SF(B): Repeatedly (even(x)) ⇒ Repeatedly(executed = B)

Note: if a spec is satisfied assuming weak fairness, it also satisfied assuming strong fairness

Encoding Strong Fairness

Process P

nat x := 0; bool y := 0; {A,B} executed
A: x := x + 1; executed := A
B: even(x) –> y := 1 - y; executed := B

Strong fairness for a task T: An infinite execution is strongly fair to task T if it satisfies the formula

SF(T): Repeatedly (T is enabled) ⇒ Repeatedly (T is executed)

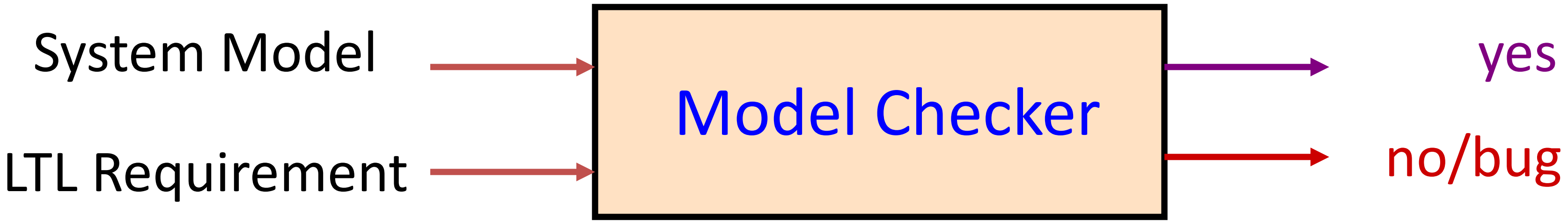
Example:

SF(B): Repeatedly (even(x)) ⇒ Repeatedly(executed = B)

Exercise: Which of the following specs are satisfied by P?

1. SF(B) ⇒ Eventually (y = 1)
2. SF(B) ⇒ Repeatedly (y = 1)
3. SF(B) ⇒ Persistently (y = 1)

Model Checking



- Performed using enumerative or symbolic search through the state-space of the program
- Success story for transitioning academic research to industrial practice
- 2007 Turing Award to Ed Clarke, Alan Emerson, and Joseph Sifakis
- Used to debug multi-core protocols, pipelined processors, device driver code, distributed algorithms in Intel, Microsoft, IBM …

Büchi Automata

A safety monitor M classifies finite executions into good and bad

Verification of safety requirements for a component C reduces to analyzing reachable states of the composition of C and M

An error execution is an execution that leads the monitor into an error state

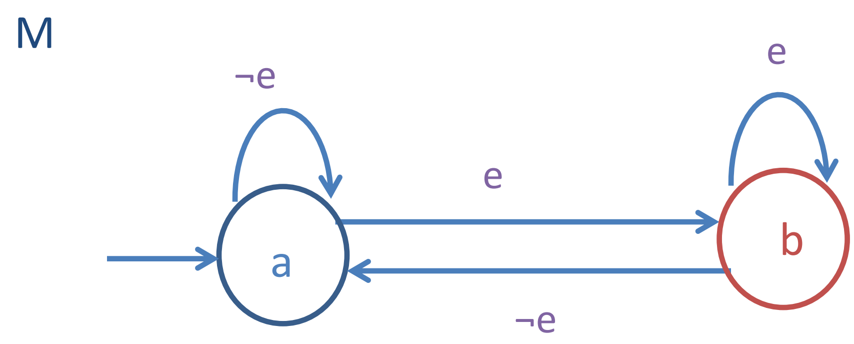
How can a monitor (aka, an automaton) classify infinite executions into good and bad?

Büchi Automata

- Theoretical model of Büchi automata proposed by Richard Büchi (1960)

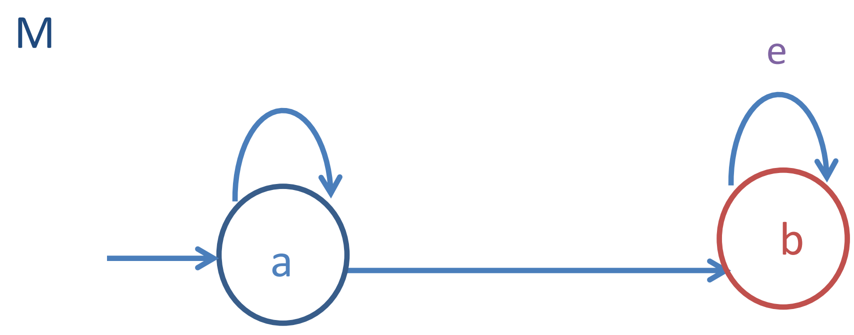
- Model checking application (1990s) using Büchi automata:
  - Automatically translate LTL formula φ to a Büchi monitor M
  - Consider the composition of system C and monitor M
  - Reachable cycles in this composite correspond to counter-examples; if no such cycle is found, system satisfies spec
  - Implemented in many model checkers (notably, SPIN)

Büchi Automaton: Example 1



- Inputs: Boolean variable e
- Of two states a and b, a is initial and b is accepting
- Given a trace ρ over e (i.e. infinite sequence of 0/1 values to e),
  there is a corresponding execution of M
- The trace ρ is accepted if accepting state appears repeatedly
- Language of M = { traces in which e is satisfied repeatedly }
- M accepts ρ iff ρ ⊨ Repeatedly e

Büchi Automaton: Example 2

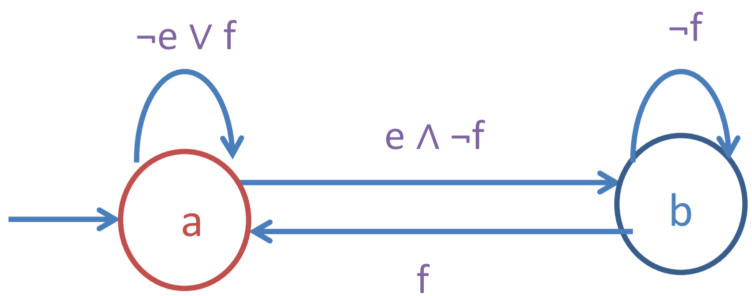


- Automaton is nondeterministic
- On a given input trace, many possible executions
- An execution is accepting if it visits accepting state repeatedly
- M accepts an input trace if there exists some accepting execution on that input
- M accepts ρ iff $\rho \vDash$ Persistently e

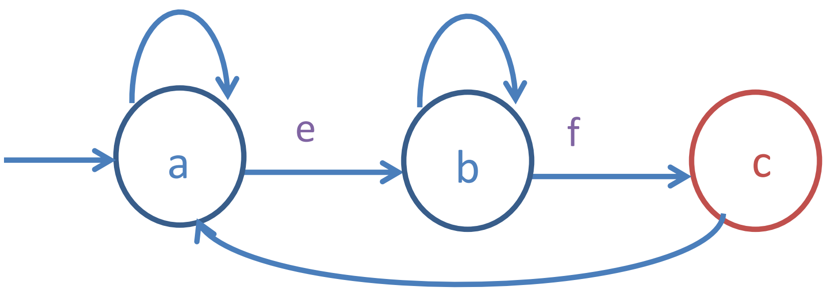
Büchi Automaton: Example 3

- Design a Büchi automaton M such that

  M accepts ρ iff $\rho \vDash$ Always (e $\Rightarrow$ Eventually f)
- Inputs: Boolean values for e and f
- In an accepting execution, every e must be followed by f



Büchi Automaton: Example 4



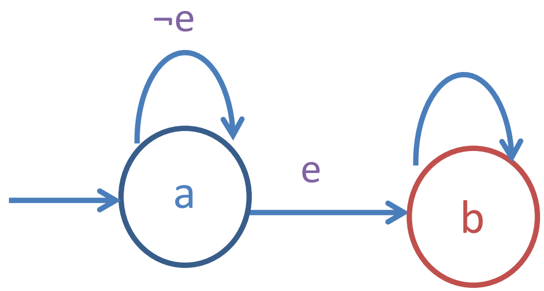
Which traces does this accept? Express it in LTL

M accepts ρ iff $\rho \vDash$ Repeatedly e ∧ Repeatedly f

Büchi Automaton M Definition

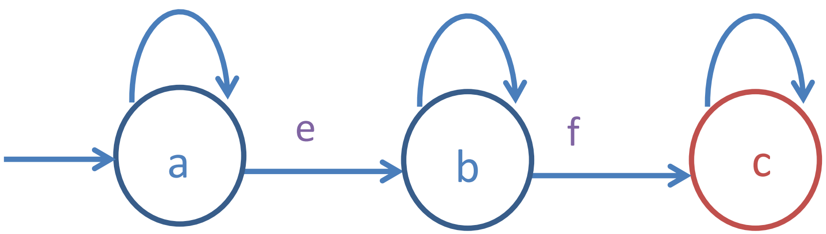
- Set of Boolean V of input variables
- Finite set Q of states
- Set $Init$ of initial states
- Set F of accepting states
- Set of edges/transitions of the form q –G–> q'
  where G is a Boolean-valued condition over V
- Given an input trace $\rho = v_1, v_2, v_3, \ldots$ over V, an accepting execution of M over ρ is an infinite sequence of states $q_0, q_1, q_2, \ldots$ where
  - q_0 is initial
  - For each i, there is an edge q_i –G–> q_{i+1} such that input v_i satisfies G
  - There are infinitely many positions i such that state q_i is in F
- M accepts input trace ρ if there is an accepting execution of M over ρ

Büchi Automata: More Examples



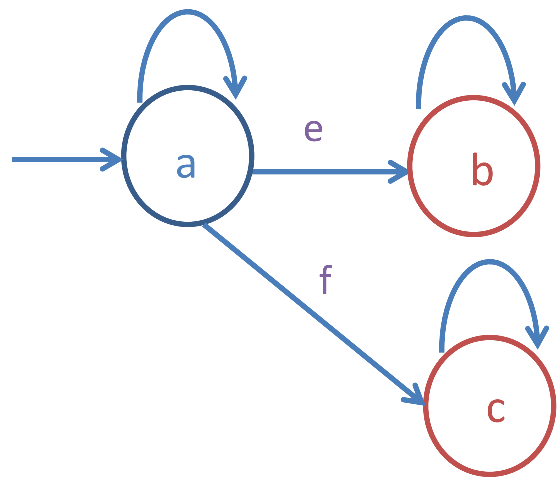
Eventually e

Büchi Automata Examples

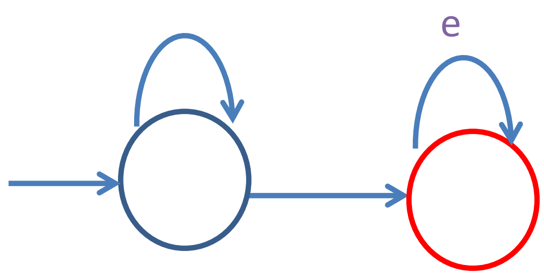


Eventually [e ∧ Next Eventually f]

Eventually e ∨ Eventually f



Nondeterministic Büchi Automaton

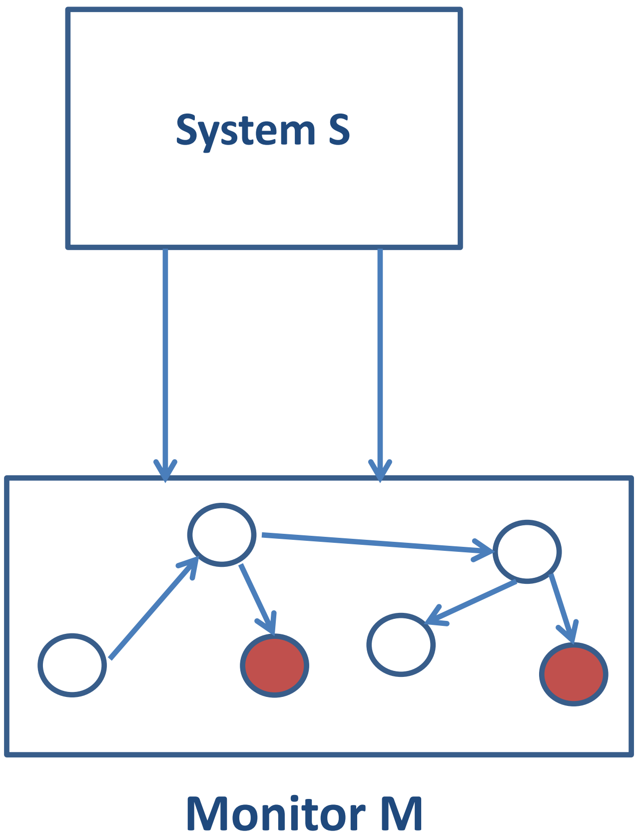


Persistently e

Can we construct an equivalent deterministic Büchi automaton?

No! Non-determinism is sometimes necessary!

Safety Monitors



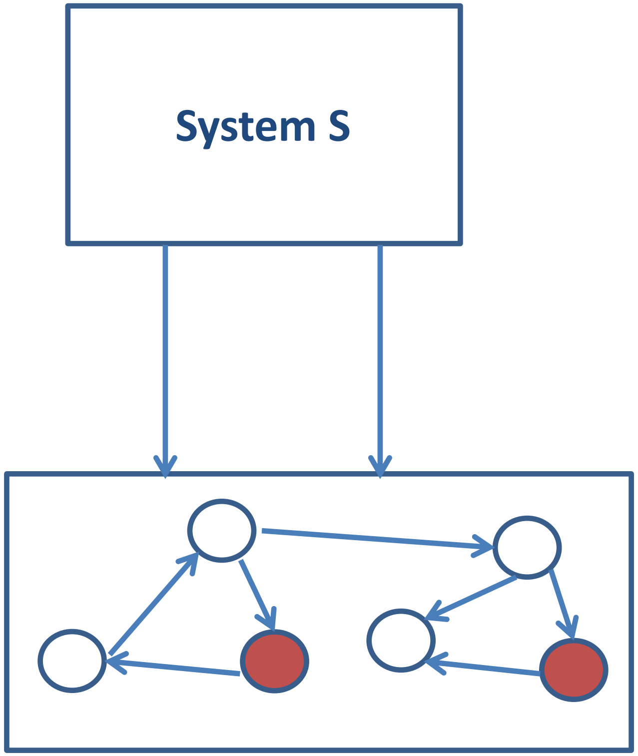
Is there an execution of **S** that makes **M** enters an error state?

M is designed so that such an execution indicates a bug!

Verification reduces to reachability

Check if an error state is reachable in composition of **S** and **M**

Büchi Monitors



Büchi Monitor M

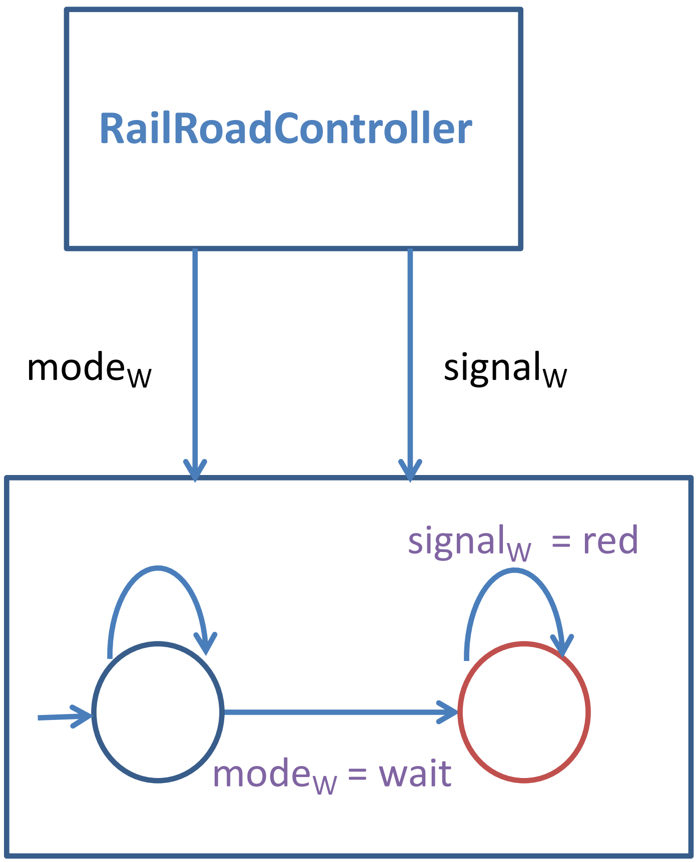
Is there an infinite execution of **S** which is accepted by **M**? (i.e., an execution in which some error state of **M** appears repeatedly?)

M is designed so that such an execution indicates a bug!

Verification reduces to search for cycles

Check if there is a reachable cycle containing an error state in the composition of **S** and **M**

Example Büchi Monitor



Büchi Monitor M

Correctness requirement:

Always [(West train is waiting) ⇒

Eventually (West signal is green)]

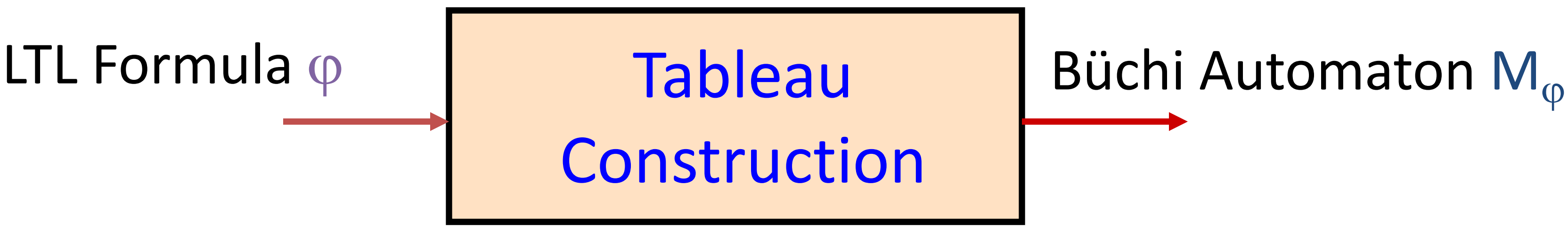
Requirement violation:

Infinite execution where, at some step, west train is waiting and in all subsequent times west signal is red

Verification:

Search for reachable cycle containing red monitor state in the combined system

From LTL to Büchi Automata



Automaton M_φ accepts exactly those traces that satisfy formula φ

To check if a system C satisfies the LTL requirement φ

- construct the Büchi automaton M_φ corresponding to φ
- search for cycles in composition of C and M_φ

Tableau Construction Example

Consider Always e ∧ Eventually f: A e ∧ E f

A state is a collection of formulas that must be satisfied

Initial state contains given formula

Formulas in a state must be consistent with rules of logical connectives: for example, if a state has $\varphi \wedge \psi$, then it must have both φ and ψ

Omega-Regular Languages

- The *language* of a Büchi automaton is the set of traces it accepts
- Such languages are called *ω-regular*

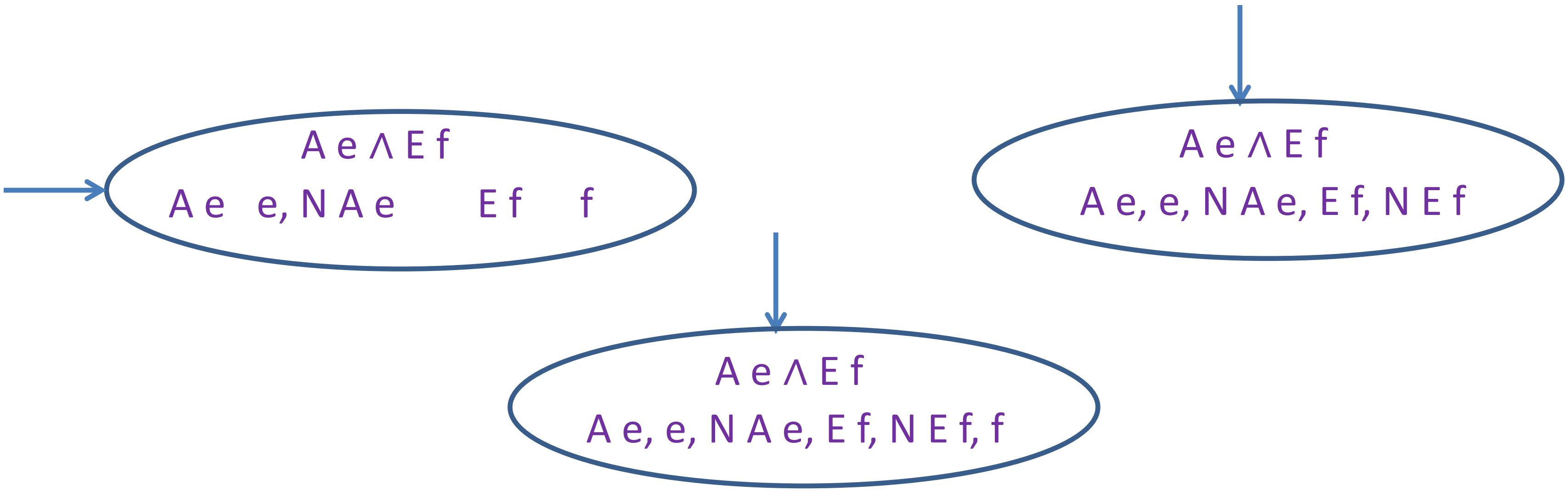
- There is a well-developed theory of ω -regular languages

- It is analogous the classical theory of regular languages (i.e., languages of finite strings of input characters accepted by finite automata)

- Relevance to us:
  Given an LTL formula φ, there is an algorithm to construct a Büchi automaton M_φ that accepts exactly the traces that satisfy φ

Tableau Construction Example

Consider Always e ∧ Eventually f: A e ∧ E f



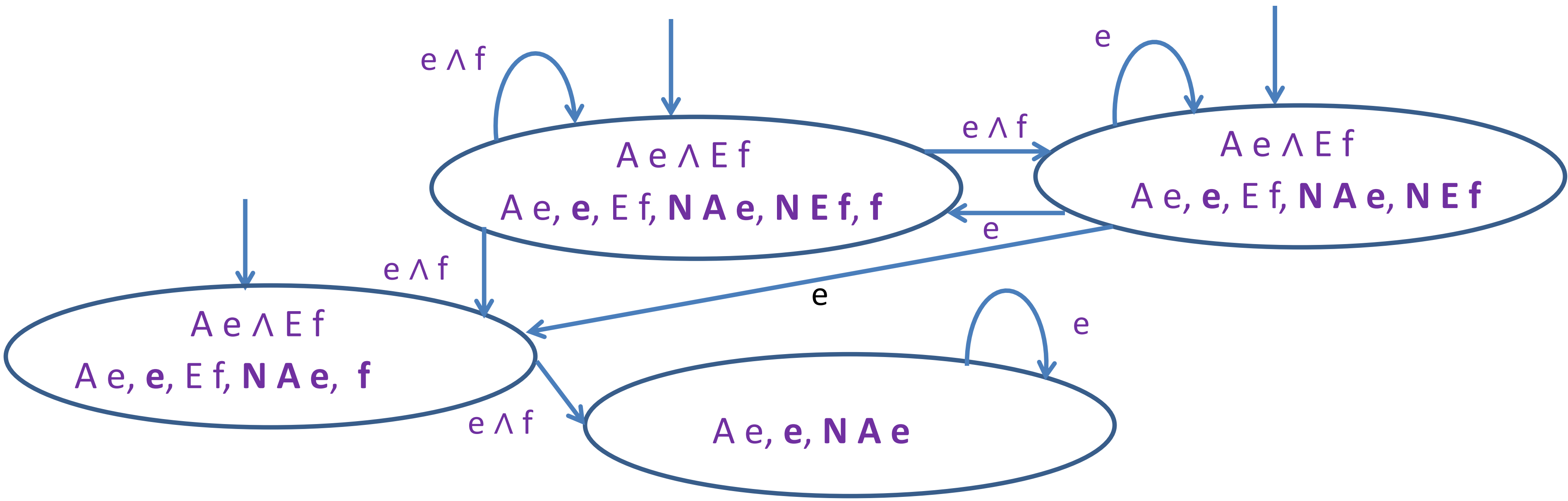
If a state has Always φ, it must have both φ and Next Always φ

If a state has Eventually φ, it must have either φ or Next Eventually φ or both

This leads to 3 cases

Tableau Construction Example

Consider Always e ∧ Eventually f: A e ∧ E f

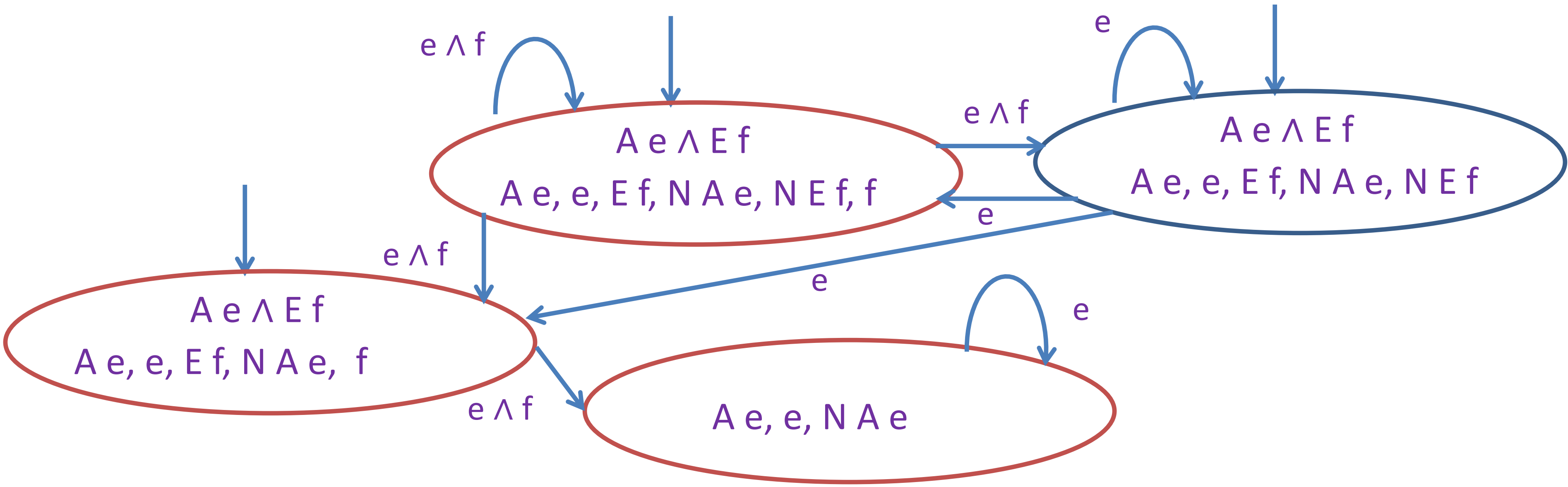


Transition Rules:

1. If a state contains Next φ then add transition to each state containing φ
2. If a state contains base formula φ, then φ must hold on outgoing transitions

Tableau Construction Example

Consider Always e ∧ Eventually f: A e ∧ E f

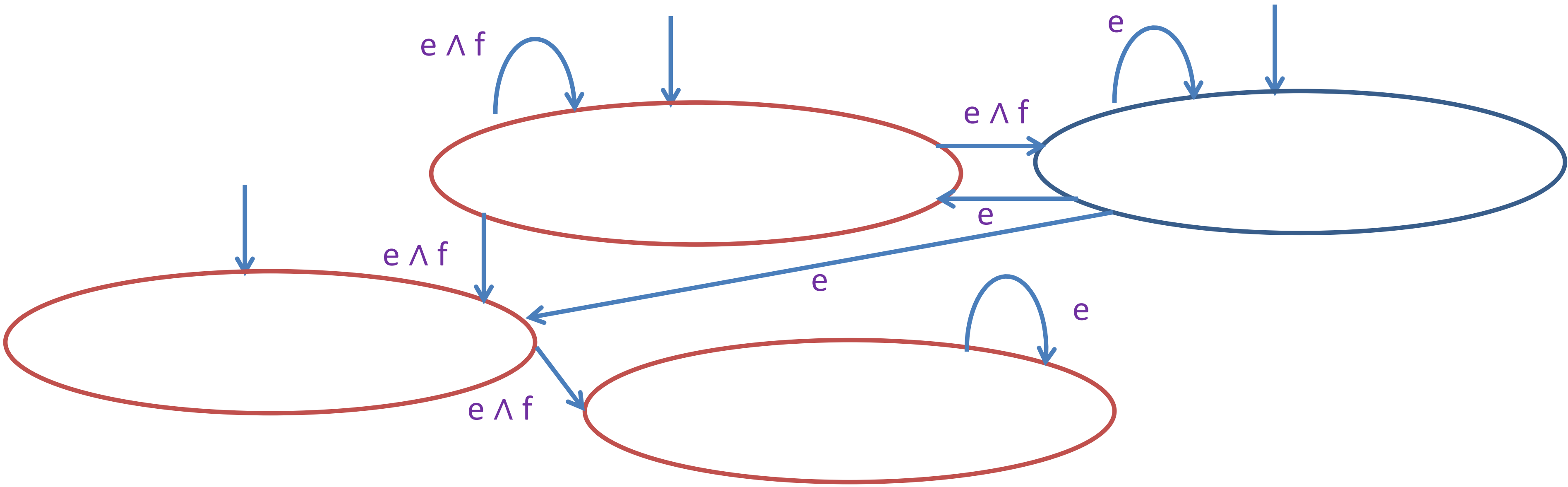


Acceptance condition: Satisfaction of eventuality should not be postponed forever

Accepting states: States that either contain f or do not contain E f

Tableau Construction Example

Consider Always e ∧ Eventually f

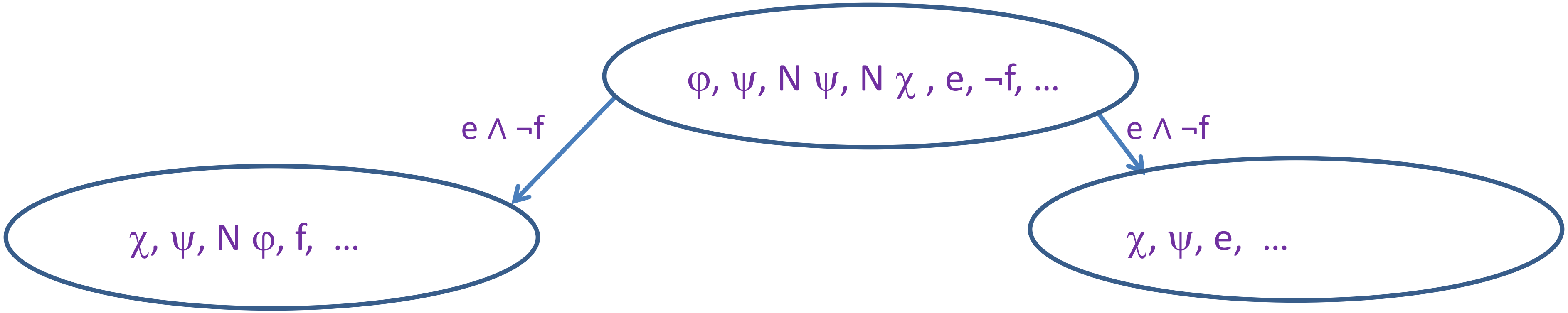


Indeed this is a correct Büchi automaton for the given formula!

Tableau Construction Overview

Automaton/tableau state: Collection of relevant LTL formulas

Intended meaning: All the formulas in a state must hold on every infinite path starting at a state



Local consistency rules ensure that for every non-atomic formula φ, the state contains additional formulas ensuring that φ holds

Transition rules ensure that

1. every atomic formula holds at current time, and
2. all Next formulas are propagated to next state

Formal Construction

Given an LTL-formula φ, define set Sub(φ), the *closure* of φ

Sub(φ) consists of formulas that are relevant to evaluation of φ:

- It contains all the subformulas of φ
- If it contains Always ψ, it also contains Next Always ψ
- If it contains Eventually ψ, it also contains Next Eventually ψ
- If it contains φ_1 U φ_2, it also contains Next (φ_1 U φ_2)

Example:

Sub(Always Eventually e ∧ Next f) =

{ Always Eventually e ∧ Next f,

Always Eventually e, Eventually e, e

Next f, f,

Next Eventually e, Next Always Eventually e }

Note: Size of Sub(φ) is linear in the size of φ

Tableau States

A state of the desired automaton is a subset of Sub(φ) that satisfies some consistency rules:

- Does not contain both a formula ψ and its negation $\neg\psi$
- Contains $\varphi_1 \wedge \varphi_2$ exactly when it contains both φ_1 and φ_2
- Contains $\varphi_1 \vee \varphi_2$ exactly when it contains at least one of φ_1 and φ_2
- If it contains Always ψ then it contains both ψ and Next Always ψ
- If it contains Eventually ψ then it contains at least one of ψ and Next Eventually ψ
- If it contains φ_1 U φ_2, it contains φ_1 or both φ_2 and Next (φ_2 U φ_2)

Note: Number of possible states is exponential in the size of φ

Example Construction

Formula φ = Eventually e ∧ Next ¬e

Sub(φ) = { E e ∧ N ¬e, E e, e, N ¬e, ¬e, N E e }

Tableau states:

q_0 = { e, N ¬e, N E e, E e, E e ∧ N ¬e }

q_1 = { e, N E e, E e }

q_2 = { e, N ¬e, E e, E e ∧ N ¬e }

q_3 = { e, E e }

q_4 = { ¬e, N ¬e, N E e, E e, E e ∧ N ¬e }

q_5 = { ¬e, N E e, E e }

q_6 = { ¬e, N ¬e }

q_7 = { ¬e }

Tableau Construction Continued

Input variables V**:** base formulas appearing in φ

States: Consistent subsets of Sub(φ)

Initial states: States that contain the formula φ

Transitions: q –G–> q' is a transition provided

- Next ψ is in q exactly when ψ is in q'
- If a base formula e is in q, then e is a conjunct in G, else ¬e is a conjunct in G

Example Construction Continued

Formula φ = Eventually e ∧ Next ¬e

Tableau states:

- q_0 = { e, N ¬e, N E e, E e, E e ∧ N ¬e }
- q_1 = { e, N E e, E e }
- q_2 = { e, N ¬e, E e, E e ∧ N ¬e }
- q_3 = { e, E e }
- q_4 = { ¬e, N ¬e, N E e, E e, E e ∧ N ¬e }
- q_5 = { ¬e, N E e, E e }
- q_6 = { ¬e, N ¬e }
- q_7 = { ¬e }

Transitions from q_0:

- q_0 –e–> q_4
- q_0 –e–> q_5

Transitions from q_1:

- q_1 –e–> q_0
- q_1 –e–> q_1
- q_1 –e–> q_2
- q_1 –e–> q_3

Transitions from q_6:

- q_6 –(¬e)–> q_6
- q_6 –(¬e)–> q_7

Tableau Construction: Acceptance

For a subformula Eventually ψ, need to ensure that satisfaction of ψ is not postponed forever.

Whenever Eventually ψ appears is in a state either ψ or Next Eventually ψ, or both, are included

Define F to be the set of tableau states that either include ψ or exclude Eventually ψ

Accepting condition: Repeatedly F

Similarly, for a subformula Always ψ,

1. Define F' to be the set of states that either include Always ψ or exclude ψ
2. A state in F' is required to appear repeatedly on an accepting run

Example Construction Continued

Formula φ = Eventually e ∧ Next ¬e

Tableau states:

- q_0 = { e, N ¬e, N E e, E e, E e ∧ N ¬e }
- q_1 = { e, N E e, E e }
- q_2 = { e, N ¬e, E e, E e ∧ N ¬e }
- q_3 = { e, E e }
- q_4 = { ¬e, N ¬e, N E e, E e, E e ∧ N ¬e }
- q_5 = { ¬e, N E e, E e }
- q_6 = { ¬e, N ¬e }
- q_7 = { ¬e }

q_1

Accepting states: { q_0, q_1, q_2, q_3, q_6, q_7 }

Initial states: { q_0, q_2, q_4 }

Transitions from q_0:

- q_0 –e–> q_4
- q_0 –e–> q_5

Transitions from q_1:

- q_1 –e–> q_0
- q_1 –e–> q_1
- q_1 –e–> q_2
- q_1 –e–> q_3

Transitions from q_6:

- q_6 –(¬e)–> q_6
- q_6 –(¬e)–> q_7

Handling Acceptance

In general, if there are multiple temporal formulas, the acceptance condition should ensure that each is satisfied

Generalized Büchi Automaton: Modest syntactic generalization

Automaton M has k accepting sets F_1, F_2, … F_k

An execution is accepting if for each j, some state in F_j appears repeatedly

Repeatedly F_1 ∧ Repeatedly F_2 ∧ … ∧ Repeatedly F_k

It is possible to *compile* a generalized Büchi automaton to a standard Büchi automaton

It is also possible to adapt cycle-detection algorithms to handle multiple accepting sets

Tableau Construction: Summary

Correctness: A trace over V satisfies a given LTL formula φ iff it is accepted by the Generalized Büchi Automaton M_φ

Complexity: Size of M_φ is 2^n, where n is the size of φ (such a blow-up is unavoidable)

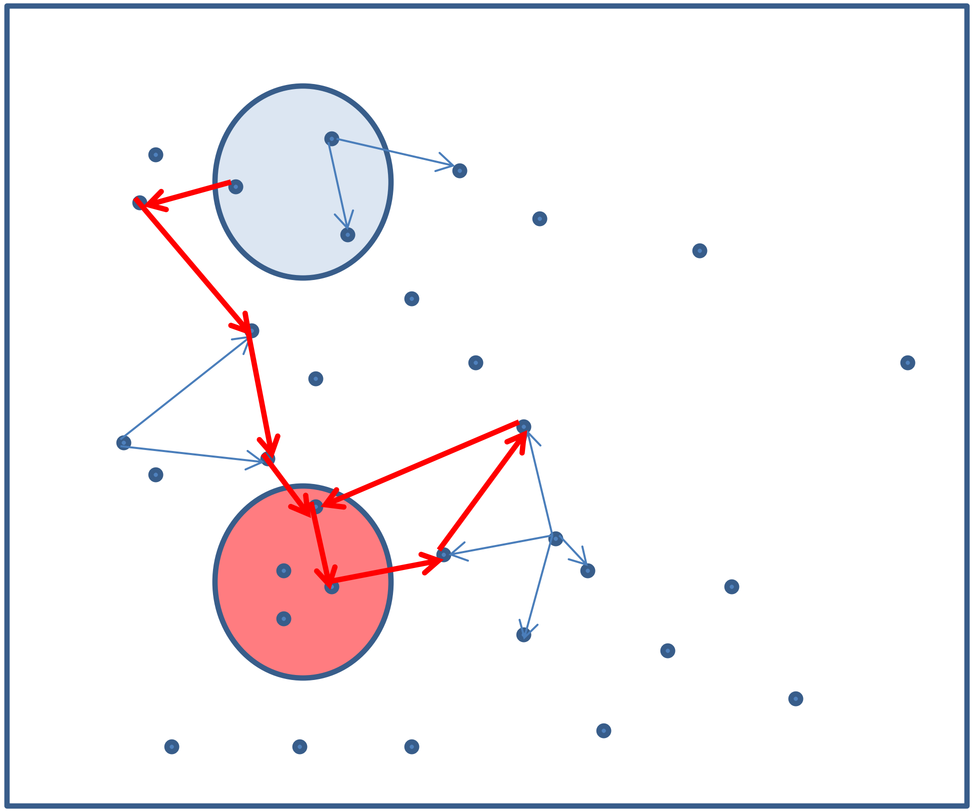
Practical implementations with a number of optimizations exist

Reachability Problem for Transition Systems

- Is there a (finite) execution from an initial state to a state satisfying φ
- Checking whether φ is an invariant of T reduces to hecking if ¬φ is reachable
- Verification techniques
  1. Proof-based: Inductive invariants
  2. Enumerative on-the-fly search (see notes)
  3. Symbolic search based on iterative image computation

Repeatable Property for Transition Systems

Transition System = (States, Initial states, Transitions)



Property φ : Subset of states

Property φ is *repeatable* if there exists an infinite execution that satisfies Repeatedly φ

Is there a state s such that

1. s is reachable
2. s satisfies φ
3. there is a cycle containing s

Repeatability Problem for Transition Systems

Is there an infinite execution along which states satisfying φ appear repeatedly?

To check whether a system C satisfies an LTL formula φ, check if property Mode is accepting is repeatable in composition of C and Büchi monitor $M_{\neg\varphi}$

Verification techniques (not covered, see Chap 5)

1. Proof-based: Ranking functions
2. Enumerative: Nested Depth-first Search
3. Symbolic search