Automated Verification of Cyber-Physical Systems

A.A. 2025/2026

Corso di Laurea Magistrale in Informatica

Basic Notions

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

General Info for This Class

- Automated Verification of Cyber-Physical Systems is an elective course for the Master Degree in Computer Science
- Lecturer: Igor Melatti
- Where to find these slides and more:
  - `https://igormelatti.github.io/aut_ver_cps/20252026/index_eng.html`
  - also on MS Teams: "DT0759: Automated Verification of Cyber-Physical Systems (2025/26)", code **ramh3r4**
- 2 classes every week, 2 hours per class

Rules for Exams

- The exam consists in either reviewing a research paper or working on a project
- Each student may choose one between the two options
- Project: perform verification of a given cyber-physical system
  - also in small teams (max 3 students)
  - each team may choose one among the ones selected by lecturer
  - or may propose one (but wait for lecturer approval!)
  - each team will have to discuss its project with slides
- Paper: read a conference or journal paper and present it with slides
  - each student may choose one among the ones selected by lecturer
  - or may propose one (but wait for lecturer approval!)
  - typically a tool paper, thus experiments reproduction is required

Automated Verification (Model Checking) Problem

- Input: a system $\mathcal{S}$ and (at least) a property φ
  - more precisely, a *model* of $\mathcal{S}$ must be provided
  - that is, $\mathcal{S}$ must be described in some suitable language
- Output:

  PASS $\mathcal{S}$ satisfies φ, i.e., $\mathcal{S} \models \varphi$
  - the system $\mathcal{S}$ is correct w.r.t. the property φ
  - mathematical certification, much better than, e.g., testing

  FAIL $\mathcal{S}$ does not satisfy φ, i.e., $\mathcal{S} \not\models \varphi$
  - the system $\mathcal{S}$ is buggy w.r.t. the property φ
  - a *counterexample* providing evidence of the error is also returned

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Model Checking vs. Other Verification Techniques

- Model checking is fully automatic
  - a model checker only needs the description of $\mathcal{S}$ and the property φ
  - "press button and go"
  - this is not true for other verification tools such as proof checkers, which require human intervention in the process
- Model checking is correct for both PASS and FAIL
  - unless the description of $\mathcal{S}$, or the property φ, are wrong
  - this is not true for other verification techniques such as testing, which only guarantees the FAIL result
  - a buggy system may pass all tests, because the error is in some *corner case*

Model Checking Shortcomings

- Only works for finite-state systems
  - typical example: you may verify a system with 3, 4 or 5 processes, but not with n processes, for a generic n
- Requires skilled personnel to write descriptions (and properties)
  - must know both the model checker language and the system
  - however, less skilled than a proof checker user
  - very few exceptions in which the model is automatically extracted from the system
  - also direct translations from digital circuits to NuSMV are available
- Very resource demanding
  - besides PASS and FAIL, also OutOfMem and OutOfTime are expected results...
  - bounded model checking: PASS is limited to execution up to a given number of steps

Model Checking Algorithms

Two main categories:

Explicit visit the graph induced by the description of $\mathcal{S}$

- very good for invariants and LTL model checking of communication protocols
- on-the-fly generation of the graph: only the reachable states are stored, the adjacency matrix is implicitly given by the description of $\mathcal{S}$
- Murphi, SPIN

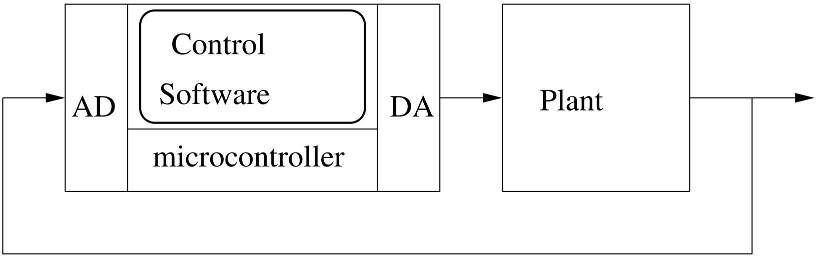
Symbolic represent sets of states and transition relations as OBDDs

- very good for LTL and CTL model checking of hardware-like systems
- all translated into a boolean formula
- also SAT tools may be used (bounded model checking)

Cyber-Physical Systems

- A Cyber-Physical System (CPS) is a system where a physical system is controlled and/or monitored by a software
- They are either partially or fully autonomous
  - we will mainly deal with fully autonomous CPSs
- Examples are everywhere:
  - Internet of Things devices
  - Unmanned Autonomous Vehicles
  - Drones
  - Medical Devices
  - Embedded Systems
  - ...

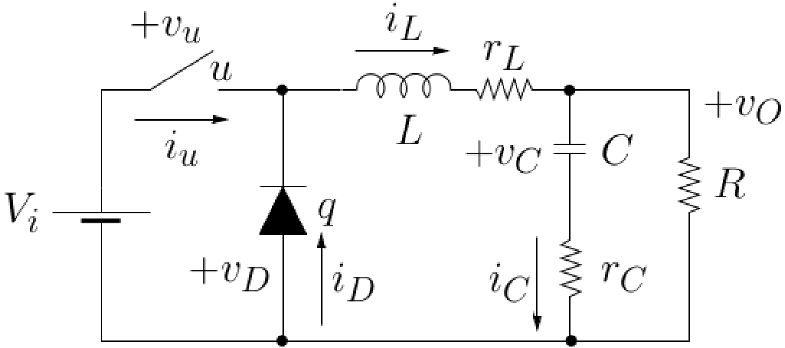
Cyber-Physical Systems with Controllers



CPSs with Controllers: Classical Examples

Buck DC/DC Converter

Buck DC/DC Converter



CPSs with Controllers: Classical Examples

Continuous time dynamics

$$\dot{i}_L = a_{1,1} i_L + a_{1,2} v_O + a_{1,3} v_D \quad (1)$$

$$\dot{v}_O = a_{2,1} i_L + a_{2,2} v_O + a_{2,3} v_D \quad (2)$$

$$q \rightarrow v_D = R_{\mathrm{on}} i_D \quad (3) \qquad \bar{q} \rightarrow v_D = R_{\mathrm{off}} i_D \quad (7)$$

$$q \rightarrow i_D \geq 0 \quad (4) \qquad \bar{q} \rightarrow v_D \leq 0 \quad (8)$$

$$u \rightarrow v_u = R_{\mathrm{on}} i_u \quad (5) \qquad \bar{u} \rightarrow v_u = R_{\mathrm{off}} i_u \quad (9)$$

$$v_D = v_u - V_{in} \quad (6) \qquad i_D = i_L - i_u \quad (10)$$

where:

- i_L, v_O are state variables
- $u \in \{0, 1\}$ is the action

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Discrete time dynamics with sampling time T

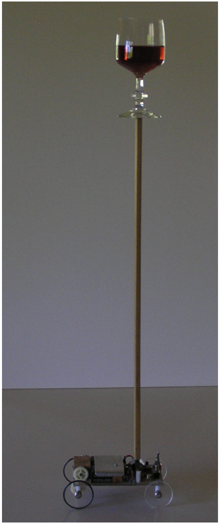
$$
\begin{aligned}
{i_L}' &= (1 + Ta_{1,1}) i_L + Ta_{1,2} v_O + Ta_{1,3} v_D & (11)\\
{v_O}' &= Ta_{2,1} i_L + (1 + Ta_{2,2}) v_O + Ta_{2,3} v_D. & (12)
\end{aligned}
$$

$$
\begin{aligned}
q &\rightarrow v_D = R_{\mathrm{on}} i_D & (13) \qquad \bar{q} &\rightarrow v_D = R_{\mathrm{off}} i_D & (17)\\
q &\rightarrow i_D \geq 0 & (14) \qquad \bar{q} &\rightarrow v_D \leq 0 & (18)\\
u &\rightarrow v_u = R_{\mathrm{on}} i_u & (15) \qquad \bar{u} &\rightarrow v_u = R_{\mathrm{off}} i_u & (19)\\
v_D &= v_u - V_{in} & (16) \qquad i_D &= i_L - i_u & (20)
\end{aligned}
$$

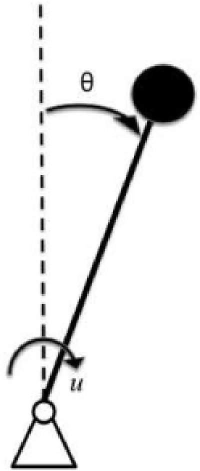
CPSs with Controllers: Classical Examples

- Goal: keep v_O in a desired safe interval
  - typically, $5 - 0.01V \leq v_O \leq 5 + 0.01V$
- Notwithstanding the input voltage V_i and the resistance R may vary in some given interval
  - typically, $R = 5 \pm 25\%\Omega$, $V_i = 15 \pm 25\% V$
- Effectively used in laptops: from battery voltage (V_i) to laptop processor voltage (v_O)

Inverted Pendulum



Inverted Pendulum



CPSs with Controllers: Classical Examples

Continuous time dynamics

$$\ddot{\theta} = \frac{g}{l} \sin\theta + \frac{1}{ml^2} F u$$

where:

- θ is the state variable
- $u \in \{0, 1\}$ is the action
- m, l, F are system parameters

CPSs with Controllers: Classical Examples

Continuous time dynamics

$$\dot{x}_1 = x_2 \tag{21}$$

$$\dot{x}_2 = \frac{g}{l}\sin x_1 + \frac{1}{ml^2}Fu \tag{22}$$

Discrete time dynamics with sampling time T

$$x_1' = x_1 + Tx_2 \tag{23}$$

$$x_2' = x_2 + T\frac{g}{l}\sin x_1 + T\frac{1}{ml^2}Fu \tag{24}$$

In This Course

To deal with cyber-physical systems:

- Probabilistic Model Checking
  - rather than "are there errors?", it is "is the error probability low enough?"
  - which entails "what is the error probability?"
  - the system is probabilistic, i.e., a Markov Chain
- Statistical Model Checking
  - rather than "are there errors?", it is "is the error probability low enough?"
  - which entails "what is the error probability?"
  - the system may be a non-probabilistic simulator
  - the answer is given with some statistical confidence
  - bridge between testing and verification

In This Course

To deal with cyber-physical systems:

- System Level Formal Verification
  - directly use a simulator instead of describing the system within the model checker
  - this will also need some background on systems simulation
  - bridge between testing and verification
- Automatic Synthesis of Controllers
  - rather than "are there errors in this system?", it is "generate a controller so that errors are avoided"

Summing up:

1. start from requirements
2. develop some (partial or final) solution
    - you may "complicate" such steps at wish
3. *verify* that the current solution fulfills the starting requirements
    - if at least one error is discovered, correct it, going to step 2
    - you may need to correct the requirements, going to step 1
    - verification may (and should) be done during the intermediate developing steps
    - if no error, deploy solution

How Verification is Performed

Method number 1: *Testing*

1. you have the actual system (or a part of it)
2. you feed it with predetermined *inputs*
3. you check if *outputs* are the expected ones
   - "expected" w.r.t. the requirements
4. if there is one output different from the expected one, then we have an error
5. you correct it and start over again
   - restarting from the "highest" point where you made the correction
   - requirements, design, code

How Verification is Performed

Method number 1 bis: *Simulation*

- two typical cases:
  - prototyping: you do not have the full code, but some simplified prototype may be built
    - feed inputs to the prototype instead of the actual software
    - especially useful to test designs (early testing)
  - you have the full code, but it is used to control/monitor some physical system (*cyber-physical systems*)
    - the simulator is for such physical system: it accepts the same inputs and provides the same outputs of the physical system
    - connect the software to such simulator as it was the real system
    - proceed as in "normal" testing by feeding inputs and observing outputs
    - you might also use a prototype for the (control/monitor) software and a simulator for the physical system for early testing

How Verification is Performed

Cyber-physical systems: why this methodology?

- Must check if they work *before* connecting to the physical part
  - or, even worse, build it
  - at least, the most common/easy errors must be ruled out
- If you have a controller for a plane, you do not directly test it on an actual plane, a simulator of the plane is used
  - only when tests on the simulator are ok you move to test on the actual plane
  - if the simulator says the plane is crashed, it is less severe than an actual plane crashing
- It is not a matter of safety only: it might also be an economical problem
  - e.g., testing on microprocessors must use some simulator before, as "writing" on silicon is expensive
  - e.g., if you are building a new airplane also basing on its controller, you must know if there are problem in the design

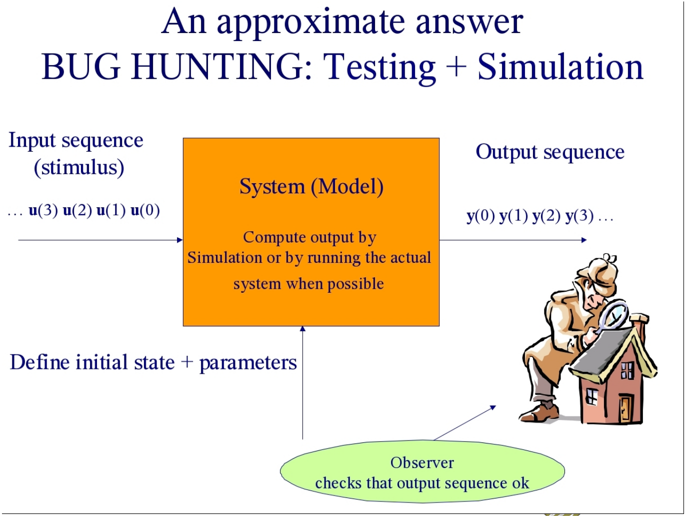
How Verification is Performed: Errors Correction

- This might not be easy: testing typically only *triggers* errors
- Then, you have to understand where the problem is and what causes it
  - requirements? architecture? design? single point in the code? an intricated flow in the code?
  - typically, you would need to re-run the test, better if in some smaller scale
  - which may cause the problem to disappear...
  - problem may arise as a consequence of some out-of-control setting: interleaving with other processes, randomization, ...
- Then, design and implement the actual correction
- In this course, we only deal with error triggering

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

How Verification is Performed



DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

How Verification is Performed

- Both testing and simulation may be performed in refined ways
- In fact, the *testing plan* (the predetermined sequence of inputs) may be computed using dedicated algorithms so that *coverage* is maximized
    - we will get back soon on this concept
- This is the most challenging and important step for such techniques

Testing and Simulation: Pro and Cons

Pro

- (Relatively) easy to implement
  - easier than the other methods we will consider here
- Largely used in industry
  - in most cases, testing and/or simulation are the *only* verification methods used

Cons

- They can prove that a system *has* errors, but cannot prove that a system *does not have* errors
- Cannot be used to prove generic formal properties
- The coverage of the "input space" is low
- Errors are frequently detected when it is too late

Testing and Simulation: Cons

They can prove that a system *has* errors, but cannot prove that a system *does not have* errors

- If an error is detected, then the system must be corrected, happy to have discovered it
- Otherwise, *we cannot conclude anything*
- That is, we cannot say that the system is error-free
- In fact, having not be able to spot errors does not imply that there are no errors

Testing and Simulation: Cons

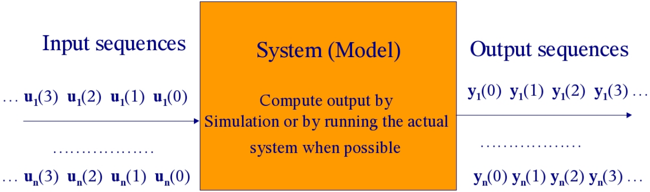
Cannot be used to prove generic formal properties

- This is a consequence of the previous slide
- As an example: in an operating system, is it true that mutual exclusion is enforced for 2 given processes?
- In order to test such a property you would have to modify the system itself
  - so that the output contains something like "propriety violated" or "'property ok"
- But even in this case, we cannot draw a formal statement on the validity of the property
- Again, not finding a violation does not imply there are no violations

Testing and Simulation: Cons

The coverage of the "input space" is low

- A successful testing phase should consider "all what may happen" to the system in a real-world environment
- This would need too much tests or simulations



- The n in the figure may easily be 10^6 and more; outputs must also be checked

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Testing and Simulation: Cons

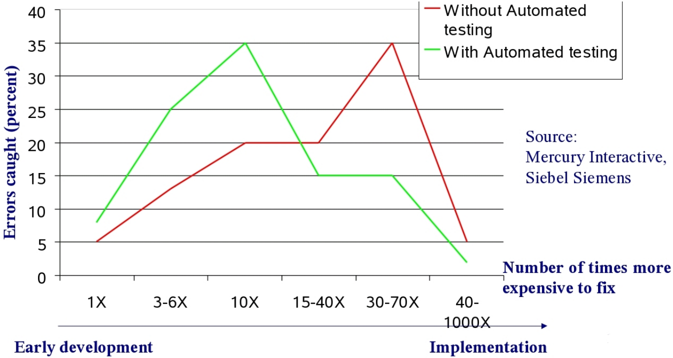
The coverage of the “input space” is low

- This also has another bad consequence
- Testing and simulation find the “easy” errors
  - the most frequent ones
  - i.e., those that are caused by many (different) input sequences
- Instead, *corner cases* usually go undetected
  - i.e., errors that are caused by a few (or even single) input sequences are usually not found

Testing and Simulation: Cons

Errors are frequently detected when it is too late

- This is a consequence of the previous point: you need many tests to get a reasonable coverage and discover possible corner cases
- The later an error is found, the more expensive the correction



Formal Verification

- To solve the above underlined problems, we should consider *all* inputs
- That is, all possible system *evolutions*
  - of course, testing and simulation only consider *some* evolutions: those "activated" by inputs chosen by the testing plan in use
- A possible way to do this is to prove a dedicated theorem, stating that the system is correct for all inputs
- For sorting, this could be done (and it is actually done in Algorithms textbooks...)
- For other cases (e.g., microprocessor design), it would be too difficult or time consuming
- Thus, techniques of *formal verification* have been developed

Formal Verification Methods

- A set of (heterogeneous) techniques which make possible the impossible
- That is, algorithms able to generate and analyze *all* system evolutions
  - so, they provide a *mathematical certification* of correctness (not achievable with testing/simulation)
  - also for generic properties, like mutual exclusion
- Actually, the problem of verifying a given system w.r.t. a given property is *undecidable*
  - the property to be verified may be: is this system always terminating?
- So, there will be some (acceptable in many cases) limitations

Is Formal Verification Useful?

- There are many techniques available for formal verification
- Applying any of these techniques is usually much more difficult than testing/simulation
  - both in terms of personnel and notions required
- So, why to do this?
- Because there are many cases in which testing/simulation simply *are not enough*
  - for both economic and safety reasons

Is Formal Verification Useful?

- Safety-critical systems: failures may affect humans
  - public transport software controllers (if an automatic pilot of an airplane has a failure...)
  - trains crossing
  - ABS for cars
  - ...
- For most of such systems, formal verification is mandatory by law
  - ESA (European Space Agency)
  - IEC (International Electrotechnical Commission)

Is Formal Verification Useful?

- Mission-critical systems: failures cause huge economic losses
  - automatic space probes
  - logistics
  - communication networks
  - microprocessors
  - ...
- Internal company regulations often make formal verification mandatory as well

Is Formal Verification Useful?

- Also for systems which are neither safety nor mission critical: there are economic motivations to use formal verification
- Using testing/simulations, errors are eventually discovered
- The problem is that they may be found *late*
  - this is a consequence of the low coverage issue
- So late, that often errors are found *after* the system has been deployed, i.e., when it is already used by its final users
  - for, e.g., a *word processor*, it is annoying, but we are somewhat used to software updates to fix bugs
  - this is not always possible or easy
    - e.g., a legacy software out of support

Is Formal Verification Useful?

- Hardware circuits: to “write” a circuit on silicon is the most expensive part of the developing process
- So, finding an error after having written the circuit entails a huge economic loss
- This also holds for other systems, when the developing process is lengthy
- In fact, finding a late error may cause going again through preceding developing phases
  - less competitivity on the market
  - for both being late and for augemented costs

Formal Verification Methodologies: a Classification

There are two macro-categories:

- *Interactive methods*
  - as the name suggests, not (fully) automatic
  - human intervention is typically required
  - in this course, we do not deal with such techniques
- *Automatic methods*
  - only human intervention is to *model* the system
- There also exist hybridations among the two categories

Interactive Methods

- Also called *proof checkers*, *proof assistants* or *high-order theorem provers*
- Tools which helps in building a mathematical proof of correctness for the given system and property
- Pros
  - virtually no limitation to the type of system and property to be verified
- Cons
  - highly skilled personnel is needed
  - both in mathematical logic and in deductive reasoning
  - needed to “help” tools in building the proof

Interactive Methods

- Used for projects with high budgets
- For which the automatic methods limitations are not acceptable
  - used, e.g., to prove correctness of microprocessor circuits or OS microkernels
- Some tools in this category (see `https://en.wikipedia.org/wiki/Proof_assistant`):
  - HOL
  - PVS
  - Coq

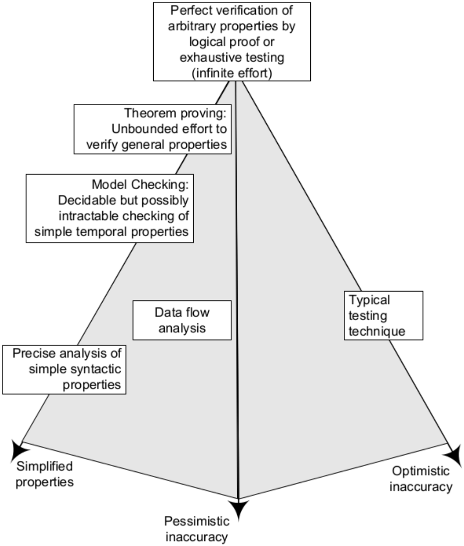
Automatic Methods

- Commonly dubbed *Model Checking*
- Model Checking software tools are called *model checkers*
- There are some tens model checkers developed; the most important ones are listed in `https://en.wikipedia.org/wiki/List_of_model_checking_tools`
- Many are freely downloadable and modifiable for research and study purposes
- Research area with many achievements in over 30 years

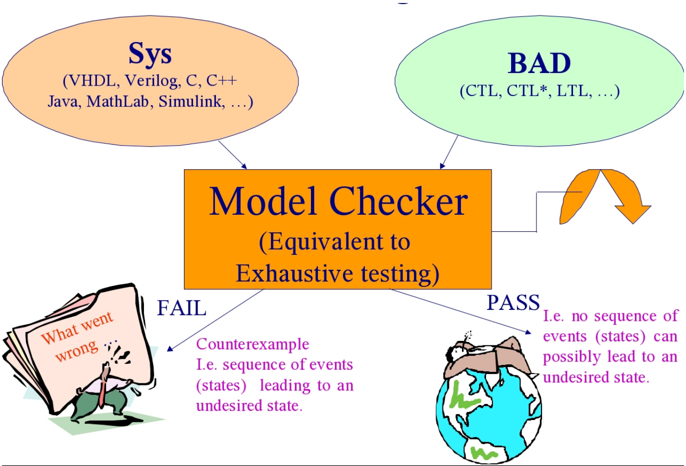
UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

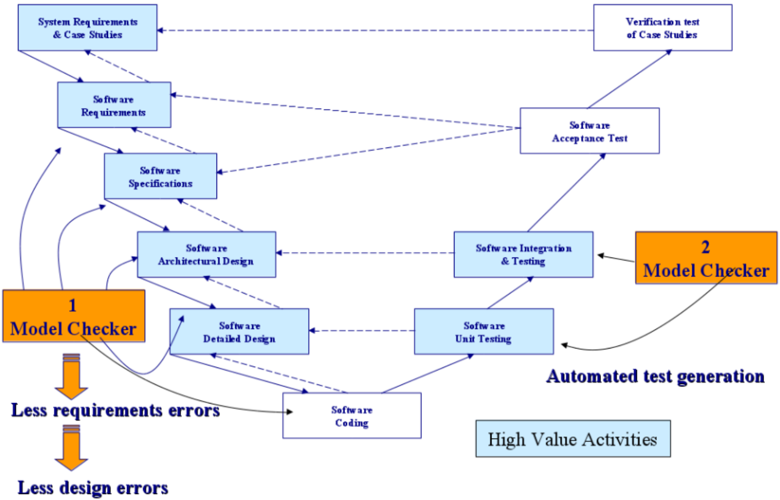
Verification Tradeoffs



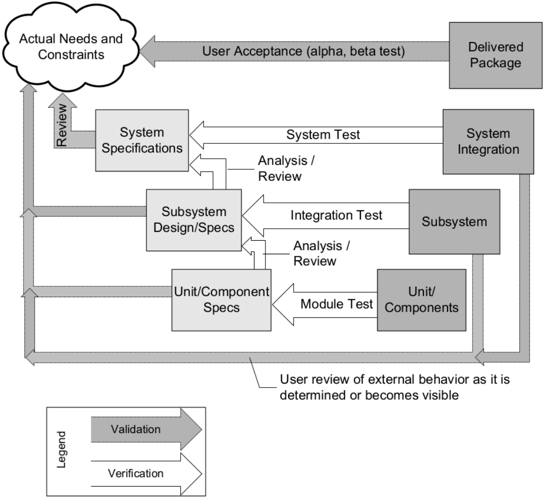
The Model Checking Dream



The Model Checking Dream



Also Keep This in Mind



Actual Model Checking

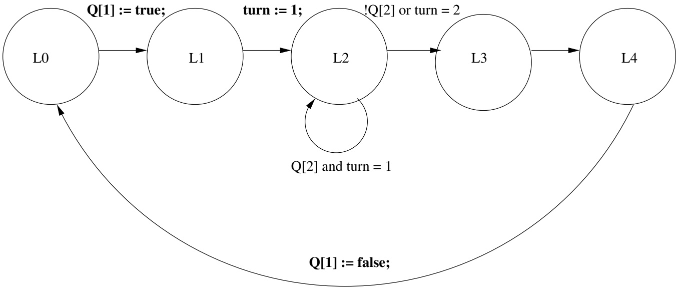
- In order to have this computationally feasible, we need a strong assumption on the system under verification (SUV)
- I.e., it must have a *finite number of states*
  - *Finite State System* (FSS)
- In this way, model checkers "simply" have to implement reachability-related algorithms on graphs
- Such finite state assumption, though strong, is applicable to many interesting systems
  - that is: many systems are actually FSSs
  - or they may be approximated as such
  - or a part of them may be approximated as such

What Is a *State*?

- There are many notions of "state" in computer science
- Model checking states are *not* the ones in UML-like state diagrams
- Model checking states are similar to operational semantics states
- That is: suppose that a system is "described" by n variables
- Then, a state is an assignment to all n variables
  - given $D_1, \ldots, D_n$ as our n variables domains, a state is $s \in \times_{i=1}^{n} D_i$

What Is a *State*: Example

- We have two identical processes accessing a shared resource
  - in the figure below, i, j denote the two processes
  - the well-known Peterson algorithm is used

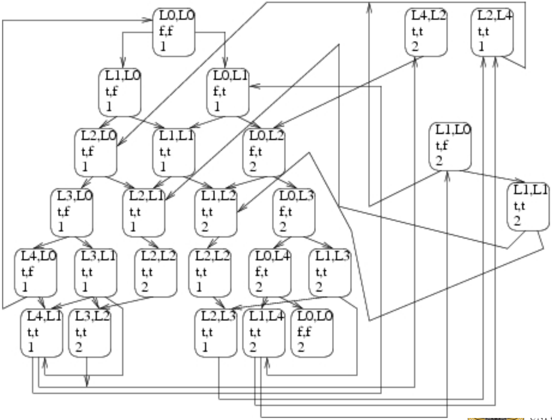


What Is a *State*: Example

- The 5 “states” in the preceding figure are actually *modalities*
- From a model checking point of view, they correspond to *multiple* (i.e., sets of) states
- To see which are the actual states, let us model this system with the following variables:
  - m_i, with $i = 1, 2$: the modality for process i
  - Q_i, with $i = 1, 2$: Q_i is a boolean which holds iff process i wants to access the shared resource
  - `turn`: shared variable

What Is a *State*: Example

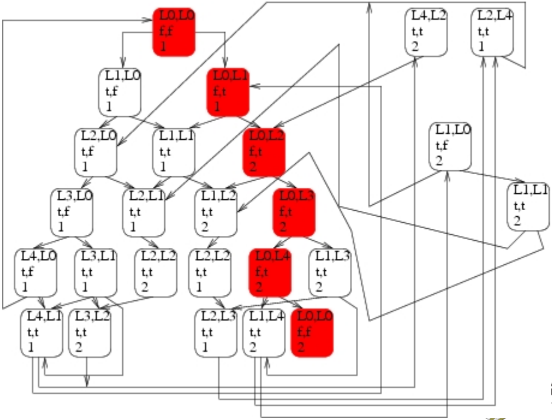
- Thus, the resulting model checking states are the following:



What Is a *State*: Example

- There are 25 *reachable states*
  - assuming state $\langle L0, L0, f, f, 1 \rangle$ as the starting one
- All *possible* states are 200
  - there are 3 variables with two possible values (the 2 variables `Q`, plus the `turn` variable) and 2 variables (`P`) with 5 possible values, thus $2^3 \times 5^2$ overall assignments
- The `L0` modality for the first process encloses 6 (reachable) states

What Is a *State*: Example



What Is a *State*: Example

- There are 25 *reachable states*
  - assuming state $\langle L0, L0, f, f, 1\rangle$ as the starting one
- All *possible* states are 200
  - there are 3 variables with two possible values (the 2 variables `Q`, plus the `turn` variable) and 2 variables (`P`) with 5 possible values, thus $2^3 \times 5^2$ overall assignments
- The `L0` modality for the first process encloses 6 (reachable) states
- No need of guards on transitions!

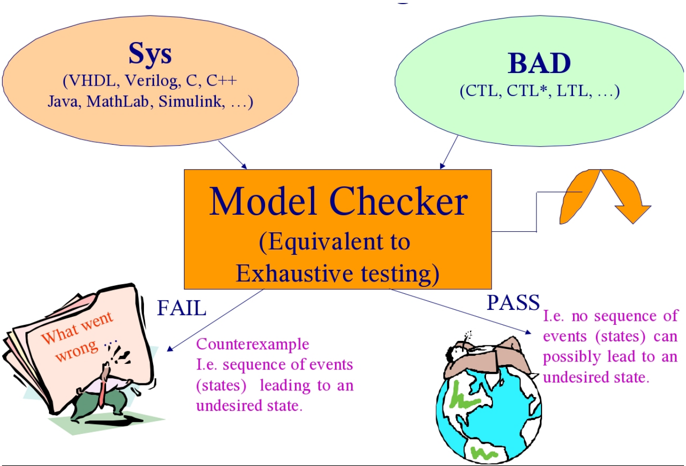
From State Diagrams to Model Checking

- The UML-like state diagram is often useful to write the model
  - as we will see, this will depend on the model checker *input language*
- It is the model checker task to extract the global (reachable) graph as seen before
- And then analyze it

Is Model Checking Important?

- ESA, NASA e IEC require most of their project to be model checked
- Important companies have dedicated laboratories for Model Checking
  - hardware: Intel, IBM, SUN, NVIDIA
  - software: IBM, SUN, Microsoft
- Many universities have research groups
  - USA: MIT, CMU, Austin, Stanford...
  - very close collaboration with companies
- The 3 “inventors” of Model Checking received Turing Award in 2007:
  - E. A. Emerson, E. M. Clarke, J. Sifakis

Model Checking Usage



Model Checking Usage

3 steps:

0. Choose the model checker M which is most suitable to the SUV $\mathcal{S}$ (and the property φ)
1. Describe $\mathcal{S}$ in the input language of M
2. Describe the property φ
3. Invoke the model checker and wait for the answer
   - OK $\Rightarrow \mathcal{S} \models \varphi$
   - FAIL $\Rightarrow$ counterexample
     - correct the error (it may happen that $\mathcal{S}$ or φ must be corrected instead...) and go back to step 3
   - OutOfMem or OutOfTime
     - adjust system parameters (or the description of $\mathcal{S}$)

Model Checking Usage

- Most used for *reactive systems*
  - always executing systems:
    - monitors: warns if something bad happens
    - controllers: avoids that something bad happens
    - services: wait for requests and serve it
  - more in general, concurrent execution of processes/threads with shared memory/messages exchange
  - errors may occur because of interactions/interleaving between different processes/threads
- Not good for standalone (1-process) programs
  - e.g., sorting an array or perform BFS of a graph
  - for such systems, testing can be complemented with theorem proving (or with manual proof derivation)
  - of course, budget must be taken into account

Model Checking: Pro and Cons

Pro

- Same guarantees of proof checking
- But requiring less “mathematics” and “computer science” knowledge

Cons

- Computational Complexity
  - causing “OutOfMem” and “OutOfTime”: *State Explosion Problem*
- You check a model of the system, not the actual system
  - though in some cases models can be automatically extracted from the system
  - the model must have a finite number of states, not always possible
- Useful only for multi-process/thread software

State Explosion Problem: Why?

- With some semplification, all Model Checking algorithms are essentially like this:
  1. Extract, from the description of the SUV $\mathcal{S}$, the *transition relation* of $\mathcal{S}$
  2. Compute the *reachable states* (*reachability*)
  3. Check if φ holds in all reachable states
- All steps may be computationally heavy, but let us focus on the reachability
  - see mutual exclusion example
- If $\mathcal{S}$ is described by n (binary) variables, then the number of reachable states is $O(2^n)$

State Explosion Problem: Why?

- Such complexity cannot be avoided in the most general case
- Theoretically speaking, (LTL) Model Checking is P-SPACE complete
  - CTL Model Checking is in P, but as we will see this does not make things better
- There are several model checking algorithms, depending on the "type" of $\mathcal{S}$
  - each checker has its "preferred" SUVs

Model Checking Algorithms

There are 3 categories:

- Explicit
- Implicit (symbolic)
- SAT-based

Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
- SAT-based

Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based

Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based
  - many problems may be theoretically rewritten as SAT, but in model checking this works pretty well also in practice
  - software model checking

Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based
  - many problems may be theoretically rewritten as SAT, but in model checking this works pretty well also in practice
  - software model checking
- Proof checker ibridations
  - not completely automatic, but better than proof checkers

What If We Use AI?

- Many powerful AI tools have been recently developed and made accessible:
  - general-purpose: ChatGPT, DeepSeek, Claude, Perplexity, ...
  - programming specific: Copilot, Llama, ...
- How they affect what we see in this course?
- We have to first consider how they can be used when developing/implementing/testing some software
  - directly generate software implementations from specifications
  - given a software, tell me if there are errors
  - ~~given a software, directly perform testing~~ AI LLMs refuse to run software
  - given a software, list some interesting test cases
  - given software specifications, output a description for some model checking tool

What If We Use AI: Software Generation

- Temptation: if it is output by AI, it is correct-by-construction
  - the verification problem simply disappears
- This is extremely far from reality
  - thus impractical for mission or safety critical software
- Finding errors in AI-generated software requires (human) developers to first understand it
- Hybrid human/AI software generation does not solve the problem

What If We Use AI: Software Analysis

- Provide the source code to AI and ask if it can spot any errors
- If an error is found, mostly good
  - AI answers may always contain errors, but a (human) developer should be able to check if the error is a false negative or not
- If an error is not found, again *no guarantee* that there are no errors
- However, asking a check to AI may be a good idea as a start of the verification

What If We Use AI: Test Cases Generation

- Provide the source code to AI and ask test cases as output
  - also specifications (for the whole software or for some parts) may be provided: white-box testing
- Again, AI answers may always contain errors
  - in this case, it may be that output test cases are not well-formed, i.e., they do not consider all inputs
  - if they are well-formed, their *coverage* (roughly, capability of finding errors, if any) could be worse than what a human tester could produce
  - especially if the methodologies explained in this course are adopted...

What If We Use AI: Model Checking Specification

- Provide the software specifications to AI and ask a specification for some model checking tool as output
- Like the generating software point, no guarantee of “correctness”
  - i.e., of representing the system correctly
  - or, if it does, of being actually usable

Murphi

- Murphi or Murφ, the simplest among "model checkers"
  - as all model checkers we will see in this course, Murphi may be freely downloaded with the source code, thus it may also be modified
  - links for download of all model checkers we will see are on the course web-page: `https://igormelatti.github.io/sw_test_val/20252026/index.html`

Murphi

- Formally, as all model checkers, Murphi needs the following input:
  1. a description of the system $\mathcal{S}$ you want to verify (i.e., the "model" you want to "check")
     - as we will see, this is essentially a Kriepke structure
  2. a property φ you want the system $\mathcal{S}$ to satisfy
- The output will be either OK or FAIL
  - if FAIL, it is possible to tell Murphi to print a *counterexample*

Murphi

- In Murphi, both the description of $\mathcal{S}$ and of φ must be written in a single text file, following a precise syntax
  - in other model checkers we will see (e.g., SPIN), this syntax has a name; but this is not the case for Murphi
  - thus, we will refer to it simply as *Murphi input language*
  - as we will see, in many points Murphi input language is similar to some imperative programming languages, especially Pascal (for statements) and C (for expressions)

Murphi

- Murphi checks that all reachable states of S satisfy all invariants
  - a state $s \in S$ is *reachable* if there exists a path in the transition graph from an initial state to s
  - that is: starting from an initial state, there exists a chain of rules, each applied to the state obtained from the preceding one, leading to s
  - this is a *safety* property

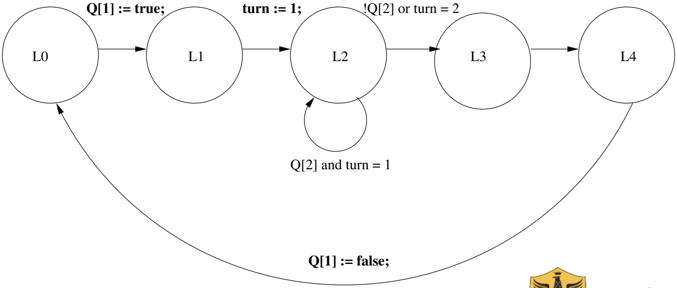
- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)

Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
           flag [0] = true;
           turn = 1;
           while (flag [1] && turn == 1) /* do nothing */;
           /* critical section */;
           flag [0] = false;
           /* remainder */;
     }
}
void P1()
{
     while (true) {
           flag [1] = true;
           turn = 0;
           while (flag [0] && turn == 0) /* do nothing */;
           /* critical section */;
           flag [1] = false;
           /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Murphi

- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)
- UML-like state diagram: this is the first process; the second may be obtained exchanging 1's with 2's and viceversa



Murphi

- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)
  - two identical processes
  - each applies Peterson protocol to access to the critical section L3
  - the first issuing the request enters L3
  - `Q` is a global variable, defined as an array of two integers
    - each process i may modify `Q[`i`]` and read `Q[`$(i+1) \mod 2$`]`
  - `turn` is another global variable, which may be both read and modified by both processes

Murphi

- Murphi description for Peterson protocol: let's start with the variables
  - of course `turn` and `Q`, but also two variables `P` for the modality ("states" in the UML-like state diagram)
  - see `01.2_peterson.no_rulesets.no_parametric.m`
  - to this aim, we define constants and types
  - the `N` constant (number of processes) is here fictious: only 2 processes, not more
  - this version of Peterson protocol only works for 2 processes
- thus, the state space is
  $S = \texttt{label\_t}^2 \times \{\texttt{true}, \texttt{false}\}^2 \times \{1, 2\}$

Variables for Murphi Model Describing Peterson Protocol

P	$v \in \{L0, L1, L2, L3, L4\}$	$v \in \{L0, L1, L2, L3, L4\}$
Q	$v \in \{true, false\}$	$v \in \{true, false\}$
turn	$v \in \{1..N\}$	

Murphi

- Hence, $|S| = 5^2 \times 2^2 \times 2 = 200$ (there are 200 possible states)
  - as a matter of comparison, the "state" L0 in the UML-like state diagram actually contains $5^1 \times 2^2 \times 2 = 40$ states...
- However, as we will see, *reachable* states are about 10 times less
- 2 initial states: `turn` may be initialized with any value in its domain
- Note that `01.2_peterson.no_rulesets.no_parametric.m` we have rules repeated 2 times in a nearly equal fashion
- This can be done in this very simple model, but in general descriptions must be *parametric*

Murphi

- If we want to check Peterson with 3 processes, currently we would have to add rules in the descipríon
  - very similar to the ones already present, only changing the index to 3
- Instead, it must be possible to only change the value of `N` from 2 to 3
- To write parametric descriptions in Murphi, rules are grouped with *rulesets*
  - an index will allow to describe the behavior of the generic process *i*
  - see `02.2_peterson.with_rulesets.no_parametric.m`, but invariant is still for two processes only

Murphi

- Finally, in `03.2_peterson.with_rulesets.parametric.m` also the invariant is parametric in `N`
  - `Exists` $x : T \;\; E(x)$ `End` is equivalent to $\bigvee_{x \in T} E(x)$
  - `Forall` $x : T \;\; E(x)$ `End` is equivalent to $\bigwedge_{x \in T} E(x)$
  - all types $T = \{x_1, \ldots, x_{|T|}\}$ are finite, thus it is a finite formula

Kripke Structures

- Let AP be a set of "atomic propositions"
  - in the sense of first-order logic: each atomic proposition is either true or false
  - tipically identified with lower case letters $p, q, \ldots$
- A *Kripke Structure* (KS) over AP is a 4-tuple $\langle S, I, R, L \rangle$
  - S is a finite set, its elements are called *states*
  - $I \subseteq S$ is a set of *initial states*
  - $R \subseteq S \times S$ is a *transition relation*
  - $L : S \to 2^{AP}$ is a *labeling function*

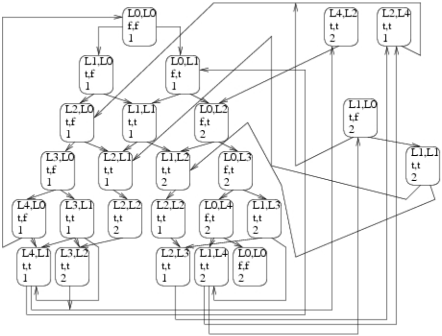
Labeled Transition Systems

- A *Labeled Transition System* (LTS) is a 4-tuple $\langle S, I, \Lambda, \delta \rangle$
  - S is a finite set of states as before
  - $I \subseteq S$ is a set of initial states as before (not always included)
  - Λ is a finite set of *labels*
  - $\delta \subseteq S \times \Lambda \times S$ is a *labeled transition relation*

Peterson's Mutual Exclusion as a Kripke Structure

- $S = \{(p_1, p_2, q_1, q_2, t) \mid p_1, p_2 \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}, q_1, q_2 \in \{0, 1\}, t \in \{1, 2\}\} = \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}^2 \times \{0, 1\}^2 \times \{1, 2\}$
- $I = \{\mathrm{L0}\}^2 \times \{0\}^2 \times \{1, 2\}$
- R: see next slide
- $AP = \{(\mathtt{P[1]} = v) \mid v \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}\} \cup \{(\mathtt{P[2]} = v) \mid v \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}\} \cup \{(\mathtt{Q[1]} = v) \mid v \in \{0, 1\}\} \cup \{(\mathtt{Q[2]} = v) \mid v \in \{0, 1\}\} \cup \{(\mathtt{turn} = v) \mid v \in \{1, 2\}\}$
    - e.g.: $L((\mathrm{L0}, \mathrm{L0}, 0, 0, 1)) = \{(\mathtt{P[1]} = \mathrm{L0}), (\mathtt{P[2]} = \mathrm{L0}), (\mathtt{Q[1]} = 0), (\mathtt{Q[2]} = 0), (\mathtt{turn} = 1)\}$

Peterson's Mutual Exclusion as a Kripke Structure



E.g.: $((\mathrm{L0}, \mathrm{L0}, 0, 0, 1), (\mathrm{L1}, \mathrm{L0}, 1, 0, 1)) \in R$, whilst
$((\mathrm{L0}, \mathrm{L0}, 0, 0, 1), (\mathrm{L2}, \mathrm{L0}, 0, 0, 1)) \notin R$

Transitions in R corresponds to arrows in the figure above

Kripke Structure vs Labeled Transition Systems

- KSs have atomic propositions on states, LTSs have labels on transitions
- In model checking, atomic propositions are mandatory
  - to specify the formula to be verified, as we will see
  - a first example was the invariant in Murphi
- Instead, it is not required to have a label on transitions
  - Murphi allows to do so, but it is optional
  - may be easily added automatically, if needed
- Labels are typically needed when:
  - we deal with macrostates, as in UML state diagrams
  - when we are describing a complex system by specifying its sub-components, so labels are used for synchronization

Total Transition Relation

- In many cases, the transition relation R is required to be *total*
- $\forall s \in S . \exists s' \in S : (s, s') \in R$
  - this of course allows also $s = s'$ (*self loop*)
- In the Peterson's example, the relation is actually total
  - Murphi allows also non-total relations, by using option `-ndl`
  - note however that not giving option `-ndl` is stronger: $\forall s \in S . \exists s' \in S : s \neq s' \wedge (s, s') \in R$
  - otherwise, if s is s.t. $\forall s' . \ s = s' \vee (s, s') \notin R$, Murphi calls s a *deadlock* state
  - that is, you cannot go anywhere, except possibly self looping on s
- By deleting any rule, we will obtain a non-total transition relation

Non-Determinism

- The transition relation is, as the name suggests, a relation
- Thus, starting from a given state, it is possible to go to many different states
  - in a deterministic system, $\forall s_1, s_2, s_3 \in S.\ (s_1, s_2) \in R \wedge (s_1, s_3) \in R \rightarrow s_2 = s_3$
  - this does not hold for KSs
- This means that, starting from state s_1, the system may *non-deterministically* go either to s_2 or to s_3
  - or many other states
- Motivations for non-determinism: modeling choices!
  - underspecified subsystems
  - unpredictable interleaving
  - interactions with an uncontrollable environment
  - ...

Some Useful Notation

- Given a KS $\mathcal{S} = \langle S, I, R, L \rangle$, we can define:
  - the *predecessor* function $\mathrm{Pre}_{\mathcal{S}} : S \to 2^S$
    - defined as $\mathrm{Pre}_{\mathcal{S}}(s) = \{s' \in S \mid (s', s) \in R\}$
    - we will write simply $\mathrm{Pre}(s)$ when $\mathcal{S}$ is understood
  - the *successor* function $\mathrm{Post} : S \to 2^S$
    - defined as $\mathrm{Post}(s) = \{s' \in S \mid (s, s') \in R\}$
- Note that, if $\mathcal{S}$ is deterministic, $\forall s \in S.\ |\mathrm{Post}(s)| \leq 1$
- Note that, if $\mathcal{S}$ is total, $\forall s \in S.\ |\mathrm{Post}(s)| \geq 1$

Paths in KSs

- A path (or *execution*) on a KS $\mathcal{S} = \langle S, I, R, L \rangle$ is a sequence $\pi = s_0 s_1 s_2 \ldots$ such that:
  - $\forall i \geq 0.\ s_i \in S$ (it is composed by states)
  - $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$ (it only uses valid transitions)
- We will denote i-th state of a path as $\pi(i) = s_i$
- Note that paths in LTSs also have actions: $\pi = s_0 a_0 s_1 a_1 \ldots$ s.t. $(s_i, a_i, s_{i+1} \in \delta)$

Paths in KSs

- The *length* of a path π is the number of states in π
  - paths can be either finite $\pi = s_0 s_1 \ldots s_n$, in which case $|\pi| = n + 1$
  - or infinite $\pi = s_0 s_1 \ldots$, in which case $|\pi| = \infty$
- We will denote the prefix of a path up to i as $\pi|_i = s_0 \ldots s_i$
  - a prefix of a path is always a finite path
- A path π is *maximal* iff one of the following holds
  - $|\pi| = \infty$
  - $|\pi| = n + 1$ and $|\mathrm{Post}(\pi(n))| = 0$
    - that is, $\forall s \in S.\ (\pi(n), s) \notin R$
    - i.e., the last state of the path has no successors
    - often called *terminal state*
- If R is total, maximal paths are always infinite
  - for many model checking algorithms, this is required

Reachability

- The set of paths of $\mathcal{S}$ starting from $s \in S$ is denoted by $\mathrm{Path}(\mathcal{S}, s) = \{\pi \mid \pi \text{ is a path in } \mathcal{S} \wedge \pi(0) = s\}$
- The set of paths of $\mathcal{S}$ is denoted by $\mathrm{Path}(\mathcal{S}) = \cup_{s \in I} \mathrm{Path}(\mathcal{S}, s)$
  - that is, they must start from an initial state
- A state $s \in S$ is *reachable* iff $\exists \pi \in \mathrm{Path}(\mathcal{S}), k < |\pi| : \ \pi(k) = s$
  - i.e., there exists a path from an initial state leading to s through valid transitions
- The set of reachable states is defined by $\mathrm{Reach}(\mathcal{S}) = \{\pi(i) \mid \pi \in \mathrm{Path}(\mathcal{S}), i < |\pi|\}$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Safety Property Verification

- Verification of *invariants*: nothing bad happens
- The property is a formula $\varphi : S \to \{0, 1\}$
  - built using boolean combinations of atomic propositions in $p \in AP$
  - i.e., the syntax is

$$\Phi ::= (\Phi) \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid p$$

- The KS $\mathcal{S}$ satisfies φ iff φ holds on all reachable states
  - $\forall s \in \mathrm{Reach}(\mathcal{S}).\ \varphi(s) = 1$
- Note that it may happen that $\varphi(s) = 0$ for some $s \in S$: never mind, if $s \notin \mathrm{Reach}(\mathcal{S})$

How to Verify a Murphi Description $\mathcal{M}$

- Theoretically, extract KS $\mathcal{S}$ and property φ from $\mathcal{M}$ as described above
  - for a given invariant I in $\mathcal{M}$, $\varphi(s) = \zeta(I, s)$ for all $s \in S$
- Then, KS $\mathcal{S}$ satisfies φ iff φ holds on all reachable states
  - $\forall s \in \mathrm{Reach}(\mathcal{S}).\ \varphi(s) = 1$
- Thus, consider KS as a graph and perform a visit
  - states are nodes, transitions are edges
- If a state e s.t. $\varphi(e) = 0$ is found, then we have an error
- Otherwise, all is ok

How to Verify a Murphi Description $\mathcal{M}$

- From a practical point of view, many optimization may be done, but let us stick to the previous scheme
- The worst case time complexity for a DFS or a BFS is $O(|V| + |E|)$ (and same for space complexity)
- For KSs, this means $O(|S| + |R|)$, thus it is linear in the size of the KS
- Is this good? NO! Because of the *state space explosion problem*
- Assuming that B bits are needed to encode each state
  - i.e., $B = \sum_{i=1}^{n} b_i$, being b_i the number of bits to encode domain D_i
- We have that $|S| = O(2^B)$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

State Space Explosion

- The "practical" input dimension is B, rather than $|S|$ or $|R|$
- Typically, for a system with N components, we have $O(N)$ variables, thus $O(B)$ encoding bits
- It is very common to verify a system with N components, and then (if N is ok) also for $N + 1$ components
  - verifying a system with a generic number N of components is a proof checker task...
- This entails an exponential increase in the size of $|S|$
- Thus we need "clever" versions of BFS/DFS

Standard BFS: No Good for Model Checking

- Assumes that all graph nodes are in RAM
- For KSs, graph nodes are states, and we know there are too many
  - state space explosion
- You also need a full representation of the graph, thus also edges must be in RAM
  - using adjacency matrices or lists does not change much
  - for real-world systems, you may easily need TB of RAM
- Even if you have all the needed RAM, there is a huge preprocessing time needed to build the graph from the Murphi specification
- Then, also BFS itself may take a long time

Murphi BFS

- We need a definition inbetween the model and the KS: NFSS (Nondeterministic Finite State System)
- $\mathcal{N} = \langle S, I, \mathrm{Post} \rangle$, plus the invariant φ
  - S is the set of states, $I \subseteq S$ the set of initial states
  - $\mathrm{Post} : S \to 2^S$ is the successor function as defined before
    - given a state s, it returns T s.t. $t \in T \to (s, t) \in R$
  - no labeling, we already have φ

Murphi BFS

- KSs and NFSSs differ on having Post instead of R
- Post may easily be defined from the Murphi specification
- Such definition is implicit, as programming code, thus avoiding to store adjacency matrices or lists
  - $t \in \mathrm{Post}(s)$ iff there is a rule $T_i \in T$ s.t. T_i guard is true in s and T_i body changes s to t
    - see above for using η and ζ
  - Essentially, if the current state is s, it is sufficient to inspect all (flattened) rules in the Murphi specification $\mathcal{M}$
    - for all guards which are enabled in s, execute the body so as to obtain t, and add t to $\mathrm{next}(s)$
  - This is done "on the fly", only for those states s which must be explored

Simple Simulation

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

Simple Simulation with Deadlock

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

Murphi Simulation

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅ ∨ Post(s_curr) = {s_curr})
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

Murphi Simulation

- Similar to testing
- If an error is found, the system is bugged
  - or the model is not faithful
  - actually, Murphi simulation is used to understand if the model itself contains errors
- If an error is not found, we cannot conclude anything
- The error state may lurk somewhere, out of reach for the random choice in `pick_a_state`

Standard BFS (Cormen-Leiserson-Rivest)

```
BFS(G, s)
for ogni vertice u ∈ V[G] − {s}
        do  color[u] ← WHITE
            d[u] ← ∞
            π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅
        do  u ← head[Q]
            for ogni v ∈ Adj[u]
                do if color[v] = WHITE
                      then  color[v] ← GRAY
                            d[v] ← d[u] + 1
                            π[v] ← u
                            ENQUEUE(Q, v)
            DEQUEUE(Q)
            color[u] ← BLACK
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Murphi BFS

```
FIFO_Queue Q;
HashTable T;

bool BFS(NFSS 𝒩, AP φ)
{
 let 𝒩 = (S, I, Post);
 foreach s in I {
  if (!φ(s))
   return false;
 }
 foreach s in I
  Enqueue(Q, s);
 foreach s in I
  HashInsert(T, s);
```

Murphi BFS

```
while (Q ≠ ∅) {
 s = Dequeue(Q);
 foreach s_next in Post(s) {
  if (!φ(s_next))
   return false;
  if (s_next is not in T) {
   Enqueue(Q, s_next);
   HashInsert(T, s_next);
  } /* if */ } /* foreach */ } /* while */
return true;
}
```

Murphi BFS

- Edges are never stored in memory
  - states are "created" when expanding the current state
  - rules are used to modify the current state so as to obtain the new one
  - at the start, you have an empty state which is modified by startstates
- (Reachable) states are stored in memory only at the end of the visit
  - inside hashtable `T`
- This is called *on-the-fly* verification
- States are marked as visited by putting them inside an hashtable
  - rather than coloring them as gray or black
  - which needs the graph to be already in memory

State Space Explosion

- State space explosion hits in the FIFO queue `Q` and in the hashtable `T`
  - and of course in running time...
- However, `Q` is not really a problem
  - it is accessed *sequentially*
  - always in the front for extraction, always in the rear for insertion
  - can be efficiently stored using disk, much more capable of RAM
- `T` is the real problem
  - random access, not suitable for a file
  - what to do?
  - before answering, let's have a look at Murphi code

Murphi Usage

- As for all *explicit* model checker, a Murphi verification has the following steps:
  0. compile Murphi source code and write a Murphi model `model.m`
  1. invoke Murphi compiler on `model.m`: this generates a file `model.cpp`
     - `mu options model.m`
     - see `mu -h` for available options
  2. invoke C++ compiler on `model.cpp`: this generates an executable file
     - `g++ -Ipath_to_include model.cpp -o model`
     - `path_to_include` is the `include` directory inside Murphi distribution
  3. invoke the executable file
     - `./model options`
     - see `./model -h` for available options

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Beyond Invariants

- Invariants represent a huge share of properties to be verified on a system
- For many systems, one may be happy with invariants only
  - "nothing bad happens", that's all folks
- However, it is not always sufficient: a non-running system of course satisfies invariants
  - no starting states, thus no reachable states...

Safety vs. Liveness

- Safety properties: something bad must never happen
  - example: in the Peterson's protocol, it must not happen that both processes are accessing the resource (L3 in the Murphi model)
- Invariants are a special case of safety properties
  - there are some safety properties which are not invariants
  - however, they can be expressed with invariants by adding variables to the Kripke Structure
  - in the following, we will consider "invariants" and "safety properties" as synonyms
- Liveness properties: something good will eventually happen
  - example: in the Peterson's protocol, both processes will eventually access the resource
  - not at the same time!
  - cannot be expressed with invariants

Safety vs. Liveness

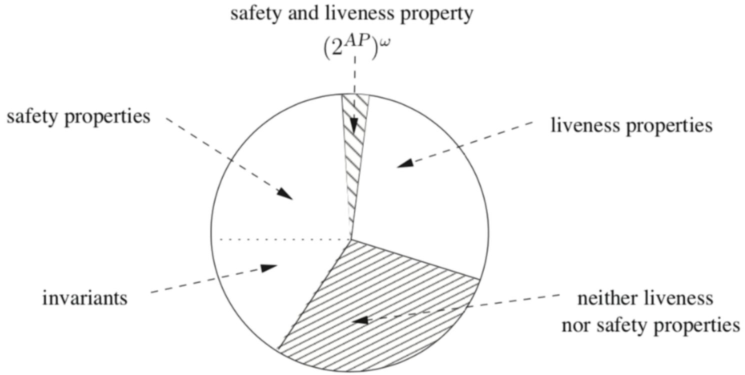
- Notation: let $\mathcal{S}$ be a KS and φ be a formula in any logic
- $\mathcal{S} \models \varphi$ is true iff φ is true in $\mathcal{S}$
  - what this means depends on the logic, as we will see
- For most properties φ, if $\mathcal{S} \not\models \varphi$ then there exists a path $\pi \in \mathrm{Path}(\mathcal{S})$ which is a *counterexample*
  - by overloading the symbol $\models$, $\pi \not\models \varphi$
- For safety properties, $|\pi| < \infty$
  - $\mathcal{S}$ arrives to an *unsafe* state and that's it
- For liveness properties, $|\pi| = \infty$
  - since $\mathcal{S}$ is finite, this implies that π contains a loop (*lasso*) in its final part

Safety vs. Liveness

- Equivalent definition for a safety formula: given a finite counterexample, every extension still contains the error
- There is one formula which is both safety and liveness: the `true` invariant
  - it cannot have a counterexample...
- There are formulas which are neither safety nor liveness
  - their counterexample is not a path
- For typically used formulas, they are either safety or liveness properties

Safety vs. Liveness: Mathematical Definition

If we identify a property by the set of its models ($\varphi = \{\sigma \mid \sigma \models \varphi\}$)



Model Checking Logics: Preliminaries

- Model Checking logics are based on the concept of *execution* of a Kripke structure $\mathcal{S}$
  - thus, on $\pi \in \mathrm{Path}$
- Often, paths are directly viewed as a sequence of atomic propositions, rather than states
  - from $\pi = s_1, s_2, \ldots$ to $AP(\pi) = L(s_1), L(s_2), \ldots$
- Focusing on executions allows to model *time*
  - time in the sense that we have something coming before of something else (in a path...)
- Trade-off between
  - logics expressiveness: interesting properties can be written
  - logics efficiency: there is an efficient model checking algorithm to compute if $\mathcal{S} \models \varphi$

Model Checking Logics: Preliminaries

- We will focus on the two leading Model Checking logics: LTL and CTL
  - with some hints on CTL*
  - LTL (Linear-time Temporal Logic) established by Pnueli in 1977
  - CTL (Computation Tree Logic) established by Clarke and Emerson in 1981
  - used for IEEE standards:
    - PSL (Property Specification Language, IEEE Standard 1850)
    - SVA (SystemVerilog Assertions, IEEE Standard 1800).
- We will see syntax and semantics of both logics
  - syntax: how a valid formula is written
  - semantics: what a valid formula "means"
  - that is, when $\mathcal{S} \models \varphi$ holds

LTL Syntax

$$\Phi ::= p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid (\Phi) \mid \mathbf{X}\Phi \mid \Phi_1 \mathbf{U} \Phi_2$$

- Other derived operators:
  - of course true, false, OR and other propositional logic connectors
  - future (or eventually): $\mathbf{F}\Phi = \mathrm{true}\ \mathbf{U}\ \Phi$
  - globally: $\mathbf{G}\Phi = \neg(\mathrm{true}\ \mathbf{U}\ \neg\Phi) = \neg\mathbf{F}\neg\Phi$
  - release: $\Phi_1\ \mathbf{R}\ \Phi_2 = \neg(\neg\Phi_1\ \mathbf{U}\ \neg\Phi_2)$
  - weak until: $\Phi_1\ \mathbf{W}\ \Phi_2 = (\Phi_1\ \mathbf{U}\ \Phi_2) \vee \mathbf{G}\Phi_1$
- Other notations:
  - next: $\mathbf{X}\Phi = \bigcirc\Phi$
  - $\mathbf{G}\Phi = \Box\Phi$
  - $\mathbf{F}\Phi = \Diamond\Phi$
- We are dropping *past operators*, thus this is *pure future LTL*

LTL Semantics

- Goal: formally defining when $\mathcal{S} \models \varphi$, being $\mathcal{S}$ a KS and φ an LTL formula
  - we say that $\mathcal{S}$ *satisfies* φ, or φ *holds in* $\mathcal{S}$
- This is true when, for all paths π of $\mathcal{S}$, π satisfies φ
  - i.e., $\forall \pi \in \mathrm{Path}(\mathcal{S}).\ \pi \models \varphi$
  - symbol $\models$ is overloaded...
- For a given π, $\pi \models \varphi$ iff $\pi, 0 \models \varphi$
- Finally, to define when $\pi, i \models \varphi$, a recursive definition over the recursive syntax of LTL is provided
  - $\pi \in \mathrm{Path}(\mathcal{S}), i \in \mathbb{N}$

LTL Semantics for $\pi, i \models \varphi$

- $\pi, i \models p$ iff $p \in L(\pi(i))$
- $\pi, i \models \Phi_1 \wedge \Phi_2$ iff $\pi, i \models \Phi_1 \wedge \pi, i \models \Phi_2$
- $\pi, i \models \neg\Phi$ iff $\pi, i \not\models \Phi$
- $\pi, i \models \mathbf{X}\Phi$ iff $\pi, i + 1 \models \Phi$
- $\pi, i \models \Phi_1 \ \mathbf{U} \ \Phi_2$ iff $\exists k \geq i : \ \pi, k \models \Phi_2 \wedge \forall i \leq j < k . \ \pi, j \models \Phi_1$

LTL Semantics for Added Operators

- It is easy to prove that:
  - $\forall \pi \in \mathrm{Path}(\mathcal{S}), i \in \mathbb{N}.\ \pi, i \models \mathrm{true}$
  - $\pi, i \models \mathbf{G}\Phi$ iff $\forall j \geq i.\ \pi, j \models \Phi$
  - $\pi, i \models \mathbf{F}\Phi$ iff $\exists j \geq i.\ \pi, j \models \Phi$
  - $\pi, i \models \Phi_1\ \mathbf{R}\ \Phi_2$ iff $\forall k \geq i.\ \pi, k \models \Phi_2 \vee \exists i \leq j < k:\ \pi, j \models \Phi_1$
    - i.e., $\forall k \geq i.\ \pi, k \not\models \Phi_2 \rightarrow \exists i \leq j < k:\ \pi, j \models \Phi_1$
    - i.e., $\forall k \geq i.\ \forall i \leq j < k.\ \pi, j \not\models \Phi_1 \rightarrow \pi, k \models \Phi_2$
  - $\pi, i \models \Phi_1\ \mathbf{W}\ \Phi_2$ iff $(\forall j \geq i.\ \pi, j \models \Phi_1) \vee (\exists k \geq i:\ \pi, k \models \Phi_2 \wedge \forall i \leq j < k.\ \pi, j \models \Phi_1)$
- For many formulas, it is silently required that paths are infinite
- That's why transition relations in KSs must be total

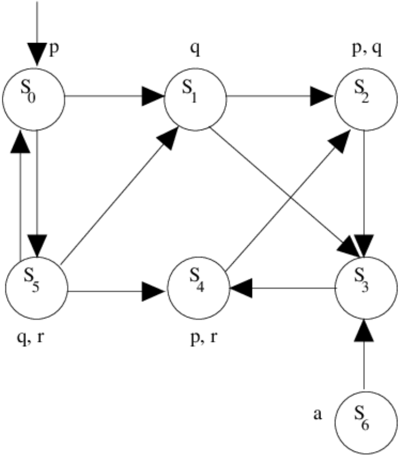
LTL Semantics: Typical Paths for Common Formulas

- For $p \in AP$, we will also consider p to be any set in $\{P \in 2^{AP} \mid p \in P\}$
  - that is, p is any subset of atomic propositions containing p
  - e.g., p may be any of $\{p\}, \{p, q\}, \{p, r, s\}$...
  - furthermore, $\bar{p} = \neg p \in \{P \in 2^{AP} \mid p \notin P\}$
    - e.g., $\bar{p}$ may be any of $\{q\}, \{q, r\}, \{r, s\}$...
  - finally, $\perp$ denotes any subset of atomic propositions
- If $\pi \models \mathbf{G}p$, then $\pi = p^\omega$
  - of course, this includes, e.g., $\pi = \{p, q\}\{p, r\}\{p\}\{p, q\}\{p\} \ldots$
  - $\pi, 3 \models \mathbf{G}p$: $\pi = \perp\perp\perp p^\omega$
- If $\pi \models \mathbf{F}p$, then $\pi = \perp^* p \perp^\omega$
- If $\pi \models p \,\mathbf{U}\, q$, then $\pi = \{p, \bar{q}\}^* q \perp^\omega$
- If $\pi \models p \,\mathbf{W}\, q$, then either $\pi = \{p, \bar{q}\}^* q \perp^\omega$ or $\pi = p^\omega$
- If $\pi \models p \,\mathbf{R}\, q$, then either $\pi = \{\bar{p}, q\}^\omega$ or $\pi = \{\bar{p}, q\}^*\{p, q\} \perp^\omega$
  - q must be kept holding till when a p appears and "releases" q...

Safety and Liveness Properties in LTL

- Given an LTL formula φ, φ is a safety formula iff $\forall \mathcal{S}.\ (\exists \pi \in \mathrm{Path}(\mathcal{S}) : \ \pi \not\models \varphi) \to \exists k : \ \pi|_k \not\models \varphi$
- Given an LTL formula φ, φ is a liveness formula iff $\forall \mathcal{S}.\ (\exists \pi \in \mathrm{Path}(\mathcal{S}) : \ \pi \not\models \varphi) \to |\pi| = \infty$
- All LTL formulas are either safety, liveness, or the AND of a safety and a liveness
  - being defined on paths, the counterexample is always a path
- Safety properties are those involving only **G**, **X**, true and atomic propositions
- Liveness are all those involving an **F** or a **U**
  - but beware of negations...
- Some formulas are both safety and liveness, like true, **G** true and so on

LTL Examples



$\mathcal{S} \models \mathbf{F}p$ since p holds in the first state

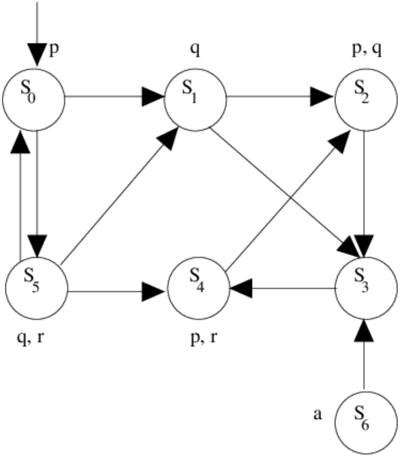
For full: let $\pi \in \mathrm{Path}(\mathcal{S})$

$\pi, 0 \models \mathbf{F}p$ with $j = 0$

recall: $\pi, i \models \mathbf{F}\Phi$ iff $\exists j \geq i.\ \pi, j \models \Phi$

$\pi, i \models p$ iff $p \in L(\pi(i))$

LTL Examples



$\mathcal{S} \not\models \mathbf{F}a$ since s_6 is not reachable from s_0

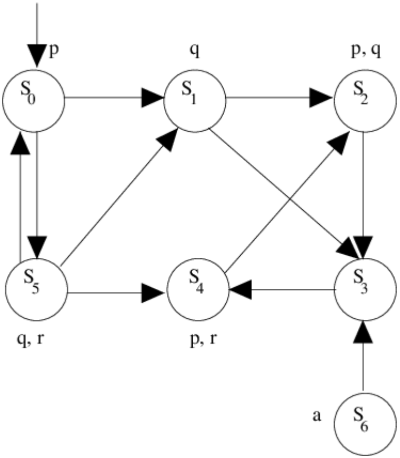
counterexample: $\pi = s_0 s_5 s_0 s_5 \ldots$

For full: $\pi, 0 \not\models \mathbf{F}a$ as, for all $j \geq 0$, $a \notin L(\pi(j))$

Counterexample is infinite, thus this is a liveness property

Any finite prefix of π is not a counterexample

LTL Examples

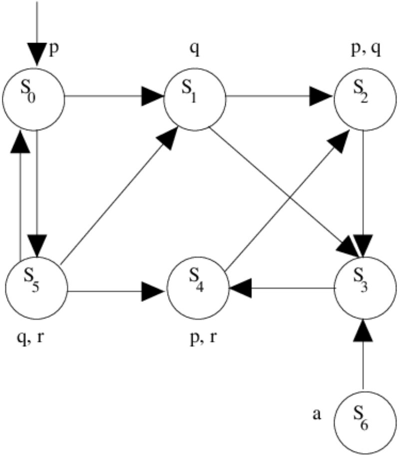


$\mathcal{S} \not\models \mathbf{G}p$ since there are many counterexamples, here is one:
$\pi = s_0 s_5 s_0 s_5 \ldots$
For full: $\pi, 0 \not\models \mathbf{G}p$ with $j = 1$

recall: $\pi, i \models \mathbf{G}\Phi$ iff $\forall j \geq i.\ \pi, j \models \Phi$
$\pi, i \models p$ iff $p \in L(\pi(i))$

Safety property, actually $\pi|_2$ is enough
Every path having $\pi|_2$ as a prefix is a counterexample

LTL Examples

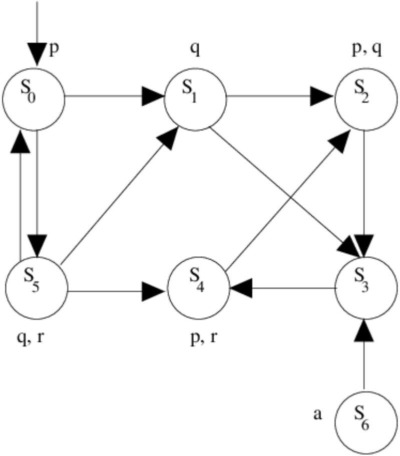


$\mathcal{S} \models \mathbf{G}\neg a$ since s_6 is not reachable from s_0

For full: let $\pi \in \mathrm{Path}(\mathcal{S})$ $\pi, 0 \models \mathbf{G}\neg a$ as the only state s with $a \in L(s)$ is s_6, which is not reachable from s_0

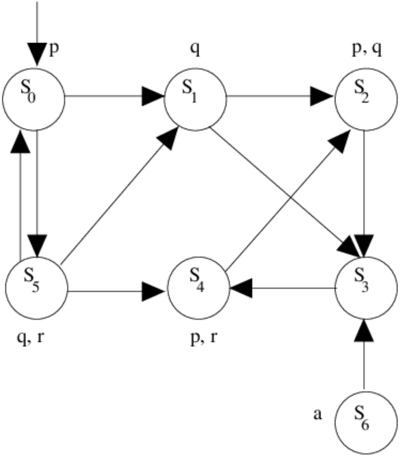
recall: $\pi \in \mathrm{Path}(\mathcal{S})$ implies $\pi(0) \in I$, thus $\pi(0) = s_0$ here

LTL Examples



$\mathcal{S} \models p \ \mathbf{U}\ q$ since $p \in L(s_0)$, $\mathrm{next}(s_0) = \{s_1, s_5\}$ and $q \in L(s_1) \wedge q \in L(s_5)$

LTL Examples

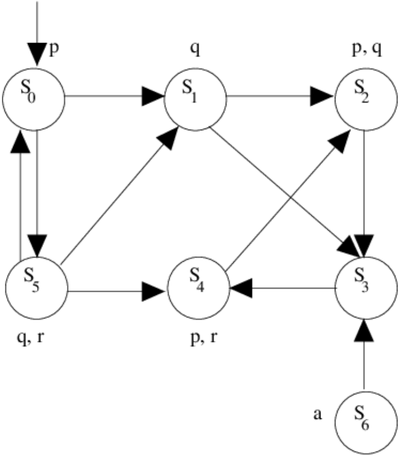


$\mathcal{S} \not\models p\ \mathbf{U}\ r$, a counterexample is $\pi = s_0 s_1 (s_2 s_3 s_4)$

Again this is a liveness formula, even if $\pi|_1$ would have been enough

In fact, you have to rule out $\{p, \bar{r}\}^\omega \ldots$

LTL Examples



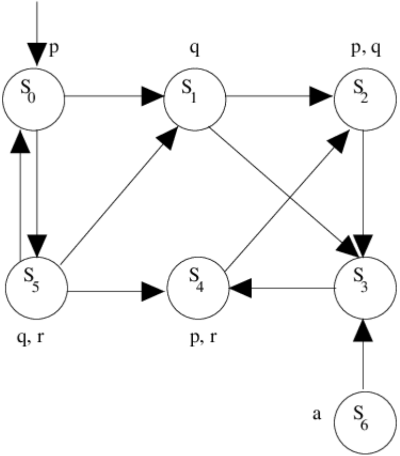
$\mathcal{S} \not\models \neg(p \ \mathbf{U} \ r)$, a counterexample is $\pi = (s_0 s_5)$

In fact, $(s_0 s_5), 0 \models p \ \mathbf{U} \ r$

Thus it may happen that $\mathcal{S} \not\models \Phi$ and $\mathcal{S} \not\models \neg(\Phi)$

Instead, it is impossible that $\mathcal{S} \models \Phi$ and $\mathcal{S} \models \neg(\Phi)$

LTL Examples



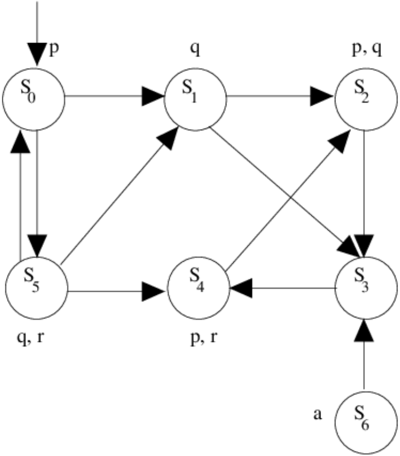
$\mathcal{S} \not\models q$, since s_0 is the only initial state and $q \notin L(s_0)$ (all paths in $\mathrm{Path}(\mathcal{S})$ must start from s_0)

$\mathcal{S} \models p$, since $p \in L(s_0)$

$\mathcal{S} \models \mathbf{X}q$, since $q \in L(s_1) \wedge q \in L(s_5)$

$\mathcal{S} \not\models \mathbf{XX}q$, since all states but s_5, s_6 are reachable in exactly 2 steps

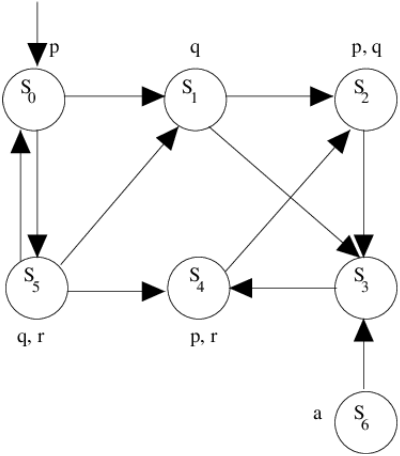
LTL Examples



$\mathcal{S} \not\models \mathbf{FG}p$, a counterexample is $\pi = s_0 s_1 (s_2 s_3 s_4)$

Again this is a liveness formula

LTL Examples

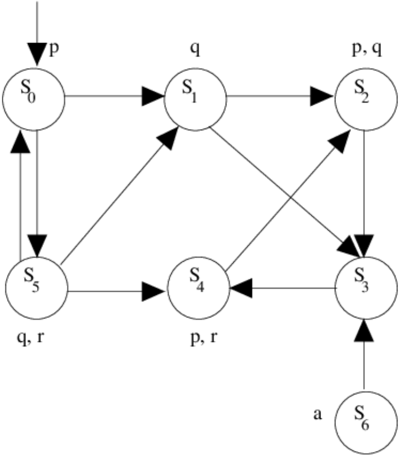


$\mathcal{S} \models \mathbf{GF}p$

All lassos are $s_0 s_5$ or $s_2 s_3 s_4$

In both such lassos, there are states in which p holds

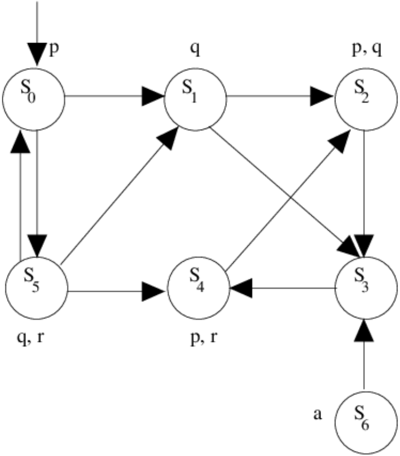
LTL Examples



$\mathcal{S} \models \mathbf{GF}p \vee \mathbf{FG}p$

Consequence of the two previous slides

LTL Examples



$\mathcal{S} \not\models \mathbf{G}(p\ \mathbf{U}\ q)$, a counterexample is $\pi = s_0 s_1 (s_2 s_3 s_4)$

$(p\ \mathbf{U}\ q)$ must hold at any reachable state

Ok in s_0, s_1, s_2, but not in s_3

LTL Non-Toy Examples

- Recall the Peterson's protocol: checking mutual exclusion is $\mathbf{G}(\neg(p \wedge q))$, being $p = \texttt{P[1]} = \texttt{L3}, q = \texttt{P[2]} = \texttt{L3}$
  - all invariants are of the form $\mathbf{G}P$, where P does not contain modal operators **X**, **U** or **F**
- Checking that both processes access to the critical section *infinitely often* is $\mathbf{GF}\ P[1] = L3 \wedge \mathbf{GF}\ P[2] = L3$
  - liveness property: no process is infinitely banned to access the critical section
- Even better: $\mathbf{G}\ (P[1] = L2 \rightarrow \mathbf{F}\ P[1] = L3)$
  - the same for the other process
  - since it is simmetric, this is actually enough

Equivalence Between LTL Properties

- Definition of equivalence between LTL properties:
  $\varphi_1 \equiv \varphi_2 \quad \text{iff} \quad \forall \mathcal{S}.\, \mathcal{S} \models \varphi_1 \Leftrightarrow \mathcal{S} \models \varphi_2$
  - equivalent: $\forall \sigma \ldots$
- Idempotency:
  - $\mathbf{FF}p \equiv \mathbf{F}p$
  - $\mathbf{GG}p \equiv \mathbf{G}p$
  - $p\ \mathbf{U}\ (p\ \mathbf{U}\ q) \equiv (p\ \mathbf{U}\ q)\ \mathbf{U}\ q \equiv p\ \mathbf{U}\ q$
- Absorption:
  - $\mathbf{GFG}p \equiv \mathbf{FG}p$
  - $\mathbf{FGF}p \equiv \mathbf{GF}p$
- Expansion (used by LTL Model Checking algorithms!):
  - $p\ \mathbf{U}\ q \equiv q \vee (p \wedge \mathbf{X}(p\ \mathbf{U}\ q))$
  - $\mathbf{F}p \equiv p \vee \mathbf{XF}p$
  - $\mathbf{G}p \equiv p \wedge \mathbf{XG}p$

CTL Syntax

$$\Phi ::= p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid (\Phi) \mid \mathbf{EX}\Phi \mid \mathbf{EG}\Phi \mid \mathbf{E}\Phi_1 \ \mathbf{U} \ \Phi_2$$

- Other derived operators (besides true, false, OR, etc):
  - $\mathbf{EF}\Phi = \mathbf{E}\mathrm{true} \ \mathbf{U} \ \Phi$
    - cannot be defined using $\mathbf{E}\neg\mathbf{G}\neg\Phi$, as this is not a CTL formula
    - actually, it is a CTL* formula (see later)
    - in fact, you cannot place a negation between **E** and the subformula
  - $\mathbf{AF}\Phi = \neg\mathbf{EG}\neg\Phi$, $\mathbf{AG}\Phi = \neg\mathbf{EF}\neg\Phi$, $\mathbf{AX}\Phi = \neg\mathbf{EX}\neg\Phi$
  - $\mathbf{A}\Phi_1 \ \mathbf{U} \ \Phi_2 = (\neg\mathbf{E}\neg\Phi_2 \ \mathbf{U} \ (\neg\Phi_1 \wedge \neg\Phi_1)) \wedge \neg\mathbf{EG}\neg\Phi_2$
  - $\Phi_1\mathbf{AU}\Phi_2 = \mathbf{A}\Phi_1\mathbf{U}\Phi_2$, $\Phi_1\mathbf{EU}\Phi_2 = \mathbf{E}\Phi_1\mathbf{U}\Phi_2$

Comparison with LTL Syntax

$$\Phi ::= \mathrm{true} \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid (\Phi) \mid \mathbf{X}\Phi \mid \Phi_1 \, \mathbf{U} \, \Phi_2$$

- Essentially, all temporal operators are preceded by either **E** or **A**
  - with some care for **U**

CTL Semantics

- Goal: formally defining when $\mathcal{S} \models \varphi$, being $\mathcal{S}$ a KS and φ a CTL formula
- This is true when, for all initial states $s \in I$ of $\mathcal{S}$, $s \models \varphi$
  - thus, CTL is made of *state* formulas
  - LTL has *path* formulas
- To define when $s \models \varphi$, a recursive definition over the recursive syntax of CTL is provided
  - no need of an additional integer as for LTL syntax

CTL Semantics for $s \models \varphi$

- $\forall s \in S.\ s \models \mathrm{true}$
- $s \models p$ iff $p \in L(s)$
- $s \models \Phi_1 \wedge \Phi_2$ iff $s \models \Phi_1 \wedge s \models \Phi_2$
- $s \models \neg\Phi$ iff $s \not\models \Phi$
- $s \models \mathbf{EX}\Phi$ iff $\exists \pi \in \mathrm{Path}(\mathcal{S}, s).\ \pi(1) \models \Phi$
- $s \models \mathbf{EG}\Phi$ iff $\exists \pi \in \mathrm{Path}(\mathcal{S}, s).\ \forall j.\ \pi(j) \models \Phi$
- $s \models \mathbf{E}\Phi_1\ \mathbf{U}\ \Phi_2$ iff
  $\exists \pi \in \mathrm{Path}(\mathcal{S}, s) \exists k :\ \pi(k) \models \Phi_2 \wedge \forall j < k.\ \pi(j) \models \Phi_1$

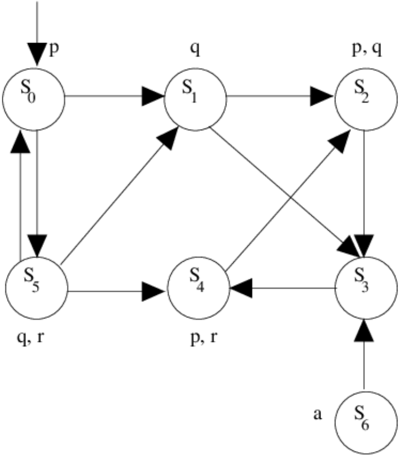
CTL Semantics for Added Operators

- It is easy to prove that:
  - $s \models \textbf{AG}\Phi$ iff $\forall \pi \in \mathrm{Path}(\mathcal{S}, s).\ \forall j.\ \pi(j) \models \Phi$
  - $s \models \textbf{AF}\Phi$ iff $\forall \pi \in \mathrm{Path}(\mathcal{S}, s).\ \exists j.\ \pi(j) \models \Phi$
  - analogously for **AU**, **AR**, **AW**
  - just replace $\forall$ with $\exists$ for **EF**, **ER**, **EW**
- Analogously to LTL, for many CTL formulas it is silently required that paths are infinite
- So again transition relations in KSs must be total

Safety and Liveness Properties in CTL

- Some CTL formulas may be neither safety nor liveness
  - being defined on states, the counterexample may be an entire computation tree
- Safety properties are those involving only **AG**, **AX**, true and atomic propositions
- Some formulas are both safety and liveness, like true, **AG** true and so on
- Liveness are formulas like **AF**, **AFAG**, **AU**
- **EF** or **EG** are neither liveness nor safety

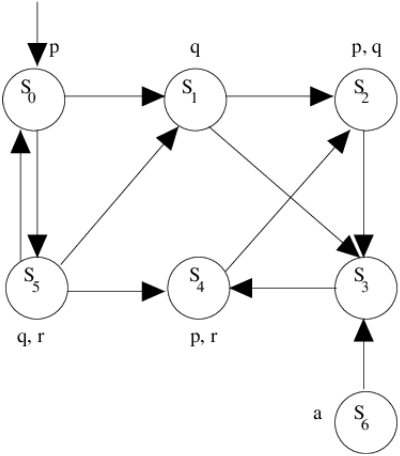
CTL Examples



$\mathcal{S} \models \mathbf{AF}p$ since p holds in the first state

For full: $s_0 \models \mathbf{F}p$ since $p \in L(s_0)$, thus, for all paths starting in s_0, p holds in the first state, so it holds eventually

CTL Examples



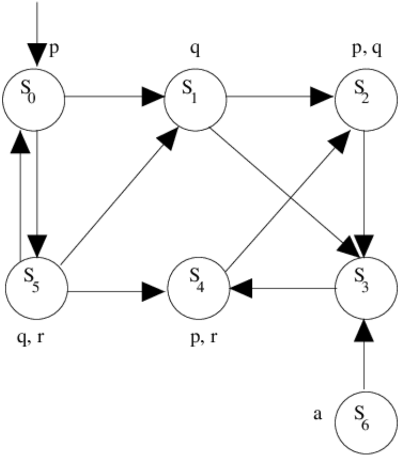
$\mathcal{S} \models \mathbf{EF}p$ for the same reason as above

If it holds for all paths, then it holds for one path

$\mathbf{AF}\Phi \rightarrow \mathbf{EF}\Phi$

The same holds for the other temporal operators $\mathbf{G}, \mathbf{U}$ etc

CTL Examples



$\mathcal{S} \not\models$ **EF**a since s_6 is not reachable

Note that the counterexample cannot be a single path

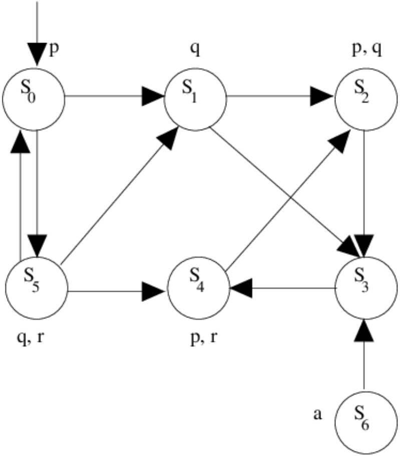
Since it would not enough to disprove existence

The full reachable graph must be provided

One could also show the tree of all paths

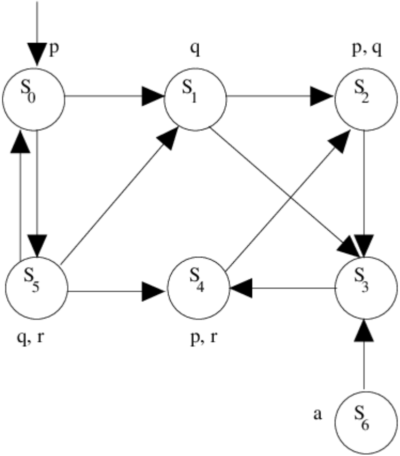
Neither safety nor liveness

CTL Examples



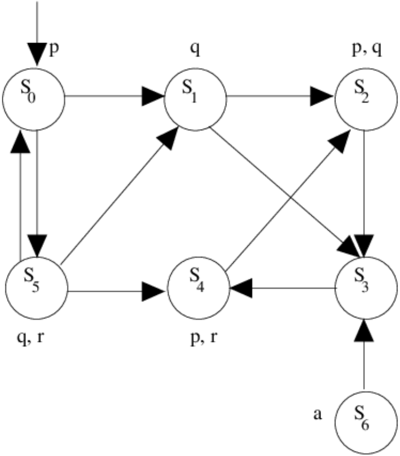
$\mathcal{S} \models \mathbf{A}(p\ \mathbf{U}\ q)$ since $p \in L(s_0)$, $\mathrm{next}(s_0) = \{s_1, s_5\}$ and $q \in L(s_1) \wedge q \in L(s_5)$

CTL Examples



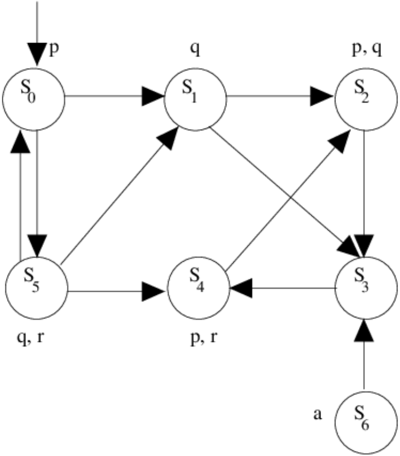
$\mathcal{S} \not\models \mathbf{A}(p \; \mathbf{U} \; r)$, a counterexample is $\pi = s_0 s_1 (s_2 s_3 s_4)$

CTL Examples



$\mathcal{S} \models \mathbf{E}(p \ \mathbf{U} \ r)$, an example is $\pi = (s_0 s_5)$

CTL Examples

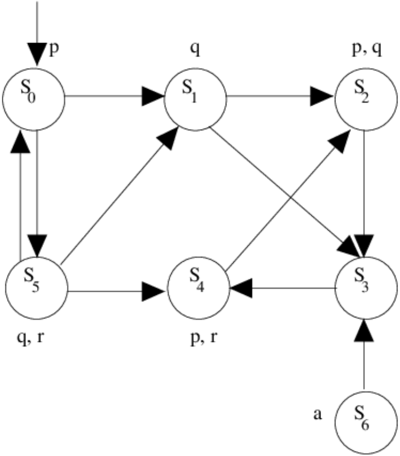


$\mathcal{S} \not\models \neg\mathbf{E}(p\ \mathbf{U}\ r)$, a counterexample is $\pi = (s_0 s_5)$

In fact, $\mathcal{S} \not\models \Phi$ iff $\mathcal{S} \models \neg(\Phi)$ whenever $|I| = 1$

In fact, the implicit for all is on initial states only, whilst it is on all paths for LTL...

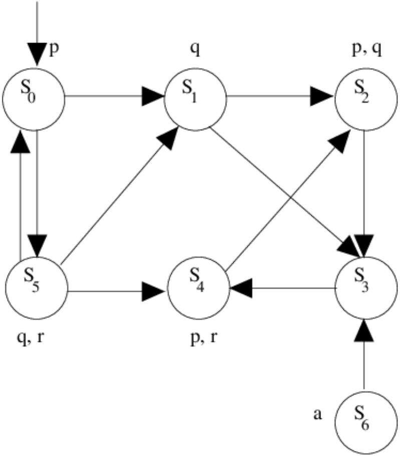
CTL Examples



$\mathcal{S} \not\models \mathbf{AFAG}p$, a counterexample is $\pi = s_0 s_1 (s_2 s_3 s_4)$

This is a liveness formula

CTL Examples

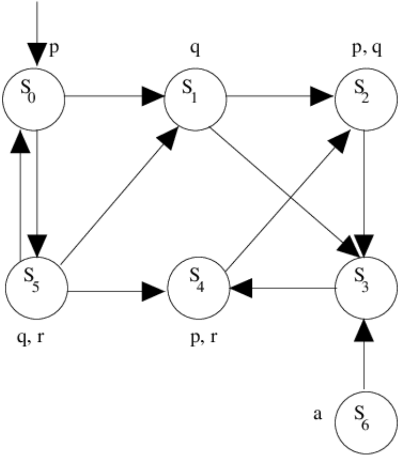


$\mathcal{S} \not\models$ **EFEG**p, a counterexample is again a computation tree

All lassos are $s_0 s_5$ or $s_2 s_3 s_4$

In both such lassos, there are states in which p does not hold

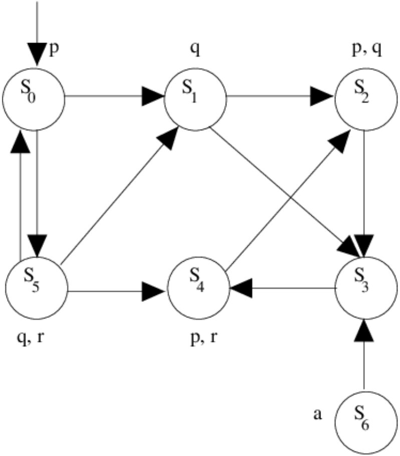
CTL Examples



$\mathcal{S} \not\models$ **AFEG**p, a counterexample is again a computation tree

Since $\mathcal{S} \not\models$ **EFEG**p...

CTL Examples



$\mathcal{S} \not\models$ **EFAG**p, a counterexample is again a computation tree

Since $\mathcal{S} \not\models$ **EFEG**p...

CTL Non-Toy Examples

- Recall the Peterson's protocol: checking mutual exclusion is $\mathbf{AG}(\neg(p \wedge q))$, being $p = \texttt{P[1]} = \texttt{L3}$, $q = \texttt{P[2]} = \texttt{L3}$
  - equivalent to LTL $\mathbf{G}p$
- It is always possible to restart:
  $\mathbf{AGEF}\ P[1] = L0 \wedge \mathbf{AGEF}\ P[2] = L0$

CTL vs. LTL: a Comparison

- Recall that $\varphi_1 \equiv \varphi_2 \quad \text{iff} \quad \forall \mathcal{S}.\, \mathcal{S} \models \varphi_1 \Leftrightarrow \mathcal{S} \models \varphi_2$
  - also holds (w.l.g.) when φ_1 is LTL and φ_2 is CTL
- Of course, some CTL formulas cannot be expressed in LTL
  - it is enough to put an **E**, since LTL always universally quantifies paths
  - so, there is not an LTL φ s.t. $\varphi \equiv \mathbf{EG}p$
    - no, $\mathbf{F}\neg p$ is not the same, why?
- So, one might think: LTL is contained in CTL
  - in the sense, for each LTL formula, there is a CTL equivalent formula
  - simply replace each temporal operator **O** with **AO**, that's it
  - let $\mathcal{T}$ be a translator doing this
  - for any LTL formula φ, $\varphi \equiv \mathcal{T}(\varphi)$
  - actually, $\mathbf{G}p \equiv \mathcal{T}(\mathbf{G}p) = \mathbf{AG}p$

CTL vs. LTL: a Comparison

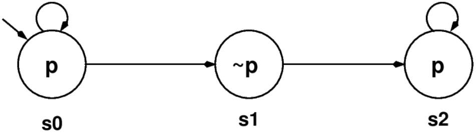
- Theorem. Let φ be an LTL formula. Then, either i) $\varphi \equiv \mathcal{T}(\varphi)$ or ii) there does not exist a CTL formula ψ s.t. $\varphi \equiv \psi$
  - idea of proof: replacing with **E** is of course not correct, and temporal operators on paths are the same
- Corollary. There exists an LTL formula φ s.t., for all CTL formulas ψ, $\varphi \not\equiv \psi$
- Proof of corollary:
  - by the theorem above and the definitions, we need to find
    1. an LTL formula φ
    2. a KS $\mathcal{S}$
  - where $\mathcal{S} \models \varphi$ and $\mathcal{S} \not\models \mathcal{T}(\varphi)$
    - viceversa is not possible

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

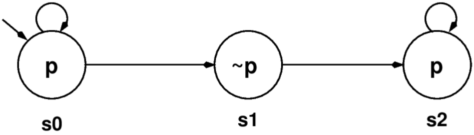
CTL vs. LTL: a Comparison

- For example, as for the LTL formula, we may take $\varphi = \mathbf{FG}p$
  - note instead that $\mathbf{GF}p \equiv \mathbf{AGAF}p$
- For example, as for the KS $\mathcal{S}$, we may take



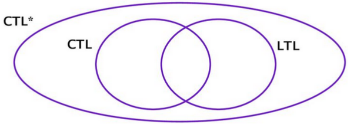
- We have that $\mathcal{S} \models \mathbf{FG}p$, but $\mathcal{S} \not\models \mathbf{AFAG}p$
- Thus, CTL requires "more" than the corresponding LTL

CTL vs. LTL: a Comparison



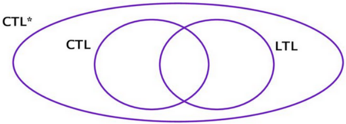
- $\mathcal{S} \not\models$ **AFAG**p means that
  $\neg(\forall \pi \in \mathrm{Path}(\mathcal{S}).\ \exists j :\ \forall \rho \in \mathrm{Path}(\mathcal{S}, \pi(j)).\ \forall k.\ p \in \rho(k))$
  $= \exists \pi \in \mathrm{Path}(\mathcal{S}).\ \forall j :\ \exists \rho \in \mathrm{Path}(\mathcal{S}, \pi(j)).\ \exists k.\ p \notin \rho(k)$
- In our $\mathcal{S}$, $\pi = s_0^\omega$: in fact, at any point of π, you may branch and go through $\neg p$ instead...
- $\mathcal{S} \models$ **FG**p means that $\forall \pi \in \mathrm{Path}(\mathcal{S}).\ \exists j :\ \forall k \geq j.\ p \in \pi(k)$
- Thus, there is not a CTL formula equivalent to **FG**p
- Furthermore, there is not an LTL formula equivalent to **AFAG**p

CTL, LTL and CTL*



- CTL* introduced in 1986 (Emerson, Halpern) to include both CTL and LTL
- No restrictions on path quantifiers to be 1-1 with temporal operators, as in CTL
- State formulas: $\Phi ::= \text{true} \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbf{A}\Psi \mid \mathbf{E}\Psi$
- Path formulas: $\Psi ::= \Phi \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi \mid \Psi_1 \mathbf{U} \Psi_2 \mid \mathbf{F}\Psi \mid \mathbf{G}\Psi$

CTL, LTL and CTL*



- The intersection between CTL and LTL is both syntactic and "semantic"
- Some formulas are both CTL and LTL in syntax: all those involving only boolean combinations of atomic propositions
- "Semantic" intersection: some LTL formulas may be expressed in CTL and vice versa, using different syntax
  - **AGAF**p and **GF**p
  - **AG**p and **G**p
  - etc

Acronyms

- Murphi stands for nothing, though it is probable that it reminds Murphi's Laws
  - "if something may fail, it will fail", i.e., $\mathbf{EF}p \rightarrow \mathbf{AF}p$
- SPIN stands for Simple Promela INterpreter
- Promela is the SPIN input language
  - Murphi input language does not have a proper name
- Promela stands for PROcess MEta LAnguage
  - as we will see, it is actually based on Operating Systems-like processes
- Also see slides at `https://spinroot.com/spin/Doc/SpinTutorial.pdf`
  - some of such slides are reused here

Structure of a Promela Model

- We recall that Murphi input language is based on:
  - global variables with finite types
    - base types are integer subranges and enumerations
    - higher types are arrays and structures
  - function and procedures
  - guarded rules and starting states (*dynamics*)
    - may call functions and procedures, in an *atomic* way
    - Pascal-like syntax: := for assignments, = for equality checks...
  - invariants

Structure of a Promela Model

- Promela instead has:
  - global variables with finite types
    - base types are integer types of the C language
    - enumerations are very limited
    - arrays and records
    - channels!
  - processes behaviour (*dynamics*)
    - possibly with arguments and local variables
  - properties to be checked:
    - assertions
    - deadlocks
    - "neverclaim" describing a BA
    - a separate tool may translate an LTL formula in the corresponding BA

Peterson Protocol in Operating Systems

Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /* do nothing */;
          /* critical section */;
          flag [0] = false;
          /* remainder */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /* do nothing */;
          /* critical section */;
          flag [1] = false;
          /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Peterson Protocol in Promela

```
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = _pid;
 (flag[1 - _pid] == 0 || turn == 1 - _pid);
 ncrit++;
 assert(ncrit == 1); /* critical section */
 ncrit--;
 flag[_pid] = 0;
 goto again
}
```

Dijkstra Protocol in Promela

```
#define p  0
#define v  1
chan sema = [0] of { bit }; /* rendez-vous */

proctype dijkstra()
{   byte count = 1; /* local variable */
    do
    :: (count == 1) -> sema!p; count = 0
    /* send 0 and blocks, unless some other
       proc is already blocked in reception */
    :: (count == 0) -> sema?v; count = 1
    /* receive 1, same as above */
    od
}
```

Dijkstra Protocol in Promela

```
proctype user()
{   do
    :: sema?p;
       /*     critical section */
       sema!v;
       /* non-critical section */
    od
}

init
{   run dijkstra();
    run user(); run user(); run user()
}
```

SPIN Simulation

Almost equal to Murphi one

```
void Make_a_run(NFSS 𝒩)
{
 let 𝒩 = ⟨S, {s0}, Post⟩;
 s_curr = s0;
 if (some assertion fail in s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (some assertion fail in s_curr))
   return with error message;
  s_curr = s_next;
 }
}
```

SPIN Verification

- Able to answer to the following questions:
  - is there a deadlock (invalid end state)?
  - are there reachable assertions which fail (safety)?
  - is a given LTL formula (safety or liveness) ok in the current system?
  - is a given neverclaim (safety or liveness) ok in the current system?
- It is possible to specify some side behaviours:
  - is sending to a full channel blocking, or the message is dropped without blocking?
- It may report unreachable code
  - Promela statements in the model which are never executed

SPIN Verification

- Similar to Murphi:
  1. the SPIN compiler (`SrcXXX/spin -a`) is invoked on `model.prm` and outputs 5 files:
     - `pan.c`, `pan.h`, `pan.m`, `pan.b`, `pan.t` (unless there are errors...)
  2. the 5 files given above are compiled with a C compiler
     - it is sufficient to compile `pan.c`, which includes all other files
     - in this way, an executable file `model` is obtained
  3. just execute `model`
     - option `--help` gives an overview of all possible options

SPIN Verification of LTL Formulas

- The former is ok for assertion or deadlock checks
- If you also have an LTL formula
  1. the SPIN compiler (`SrcXXX/spin -F`) is invoked on `model.ltl` and outputs a neverclaim on the standard output
     - `model.ltl` must be a text file with only 1 line
     - file extensions does not matter
     - syntax for the formula: **G** is `[]`, **F** is `<>`, **U** is `U`
     - atomic propositions must be identifiers
  2. append the neverclaim to the promela file
  3. define the identifiers used as atomic proposition by `#defines` in the promela file
  4. go on as before
- If you use the graphical GUI, it is much easier: such steps are automatically performed

Standard Recursive DFS

```
HashTable Visited = ∅;

DFS(graph G = (V, E), node v)
{
  Visited := Visited ∪ v;
  foreach v′ ∈ V t.c. (v, v′) ∈ E {
    if (v′ ∉ Visited)
      DFS(G, v′);
  }
}
```

Iterative DFS Easy Version

```
DFS(graph G = (V, E))
{
  s := init;
  push(s, 1);
  while (stack ≠ ∅) {
    (s, i) := top();
    increment i on the top of the stack;
    if (s ∉ Visited) {
      Visited := Visited ∪ s;
      let S' = {s' | (s, s') ∈ E};
      if (|S'| >= i) {
        s := i-th element in S';
        push(s, 1);
      }
      else pop();
    }
    else pop();
} }
```

Iterative DFS

```
DFS(graph G = (V, E))
{
  s := init; i := 1; depth := 0;
  push(s, 1);
Down:
  if (s ∈ Visited)
    goto Up;
  Visited := Visited ∪ s;
  let S' = {s' | (s, s') ∈ E};
  if (|S'| >= i) {
    s := i-th element in S';
    increment i on the top of the stack;
    push(s, 1);
    depth := depth + 1;
    goto Down;
  }
```

Iterative DFS

```
Up:
  (s, i) := pop();
  depth := depth - 1;
  if (depth > 0)
    goto Down;
}
```

Partial Order Reduction

- POR does not try to use less memory to save the same states: it tries to save less states
  - while retaining correctness, of course
  - some states are "useless" and need not to be explored (and saved)
  - also saves in computation time, of course
- Similar to Murphi symmetry for the goal, but different in use and algorithm
  - use: Murphi modeler must specify which parts of the model are symmetric
  - in SPIN, POR is directly applied without the modeler being aware of it
  - though it is possible to disable it

CTL (and LTL) Model Checking

- We saw the theoretical algorithm for CTL model checking
  - we said it was not effective, as it required S and R to be in RAM
- Actually, there are methodologies which are able to fit S and R in RAM, also for industrial-sized models
- The "father" of the model checkers using such technologies is SMV
  - Symbolic Model Verifier
  - it has then been refactored as NuSMV
- This set of techniques is referred to as *symbolic model checking*
  - Murphi and SPIN style is dubbed *explicit model checking*

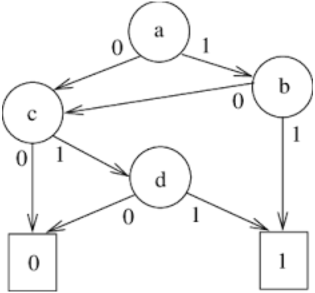
CTL (and LTL) Model Checking

- In order to understand how symbolic model checking works, we need some preliminaries
- ROBDDs
  - needed to actually fit S and R in RAM
- μ-calculus
  - together with fixpoint computation
  - extension of λ-calculus
  - needed to efficiently implement CTL and LTL model checking using ROBDDs

ROBDD

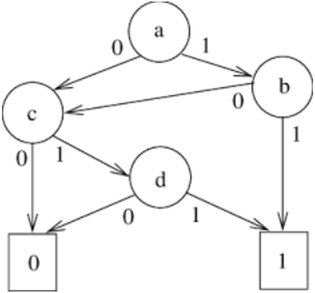
- Reduced Ordered (Complemented Edges) Binary Decision Diagrams
  - sometimes called simply OBDDs, and even BDDs
  - here we stick to the precise notation, by also outlining the differences
- Let us start with the basis: BDD
- A BDD is a data structure representing a boolean function
  - of course, OBDDs and ROBDDs are data structures as well
  - we will define them in the following

Binary Decision Diagrams



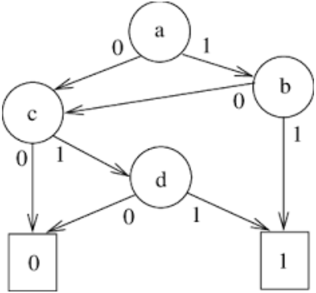
Represented function: $f(a,b,c,d) = ab + \bar{a}cd + a\bar{b}cd$

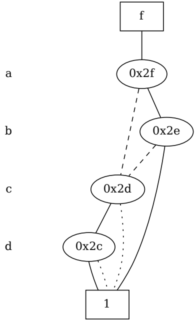
OBDDs



Supposing that $V = \mathcal{V}$, a possible ordering is:
$\mathrm{ord}(a) = 1, \mathrm{ord}(b) = 2, \mathrm{ord}(c) = 3, \mathrm{ord}(d) = 4$
If b were connected to d instead of c, also:
$\mathrm{ord}(a) = 1, \mathrm{ord}(b) = 3, \mathrm{ord}(c) = 2, \mathrm{ord}(d) = 4$

COBDDs





Represented function: $f(a,b,c,d) = ab + \bar{a}cd + a\bar{b}cd$

straight: then, dashed: else, dotted: complemented else

NuSMV Input Language

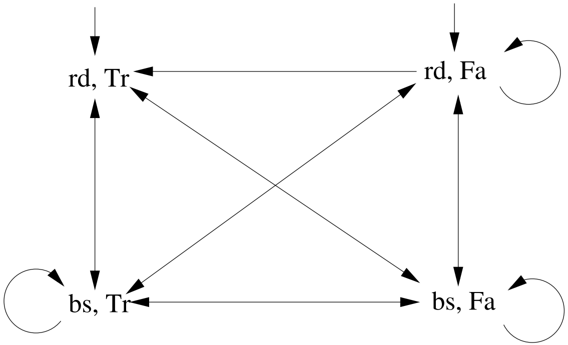
Taken from examples/smv-dist/short.smv

```
MODULE main
VAR
  request : {Tr, Fa}; -- same as saying boolean
                      -- (stand for True and False)
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : {ready,busy};
                 esac;
SPEC
  AG((request = Tr) -> AF state = busy)
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

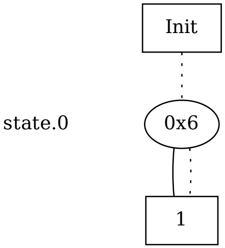
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Automata for `short.smv`: *I* and *R*



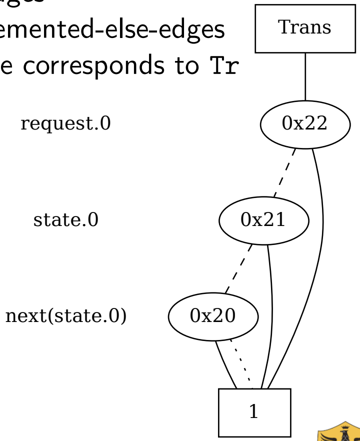
OBDDs for `short.smv`: I

Straight lines are then-edges
Dashed lines are else-edges
Dotted lines are complemented-else-edges



OBDDs for `short.smv`: R

Straight lines are then-edges
Dashed lines are else-edges
Dotted lines are complemented-else-edges
`request.0` “false” edge corresponds to `Tr`



NuSMV Input Language

```
MODULE main
VAR
  request : {Tr, Fa};
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : {ready,busy};
                 esac;
SPEC
  AG((request = Tr) -> AF state = busy)
```

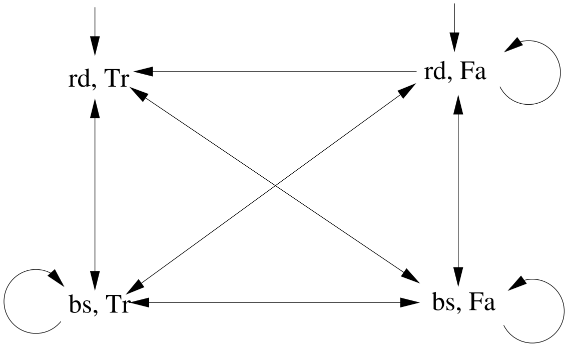
UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

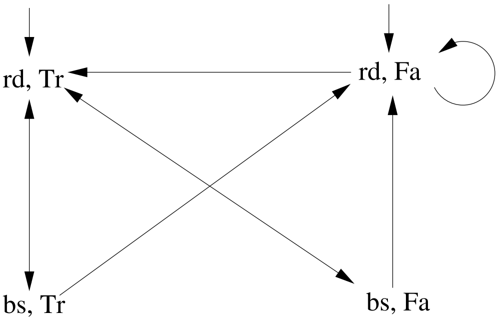
NuSMV Input Language

```
MODULE main
VAR
  request : {Tr, Fa};
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : ready;
                 esac;
SPEC
  AG((request = Tr) -> AF state = busy)
```

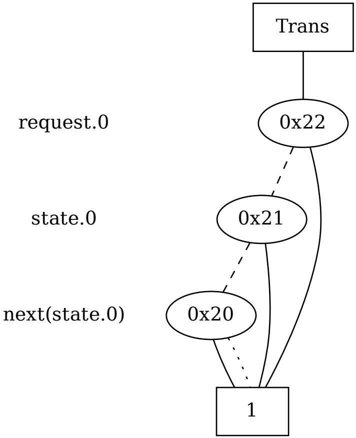
Automata for `short.smv`: I and R



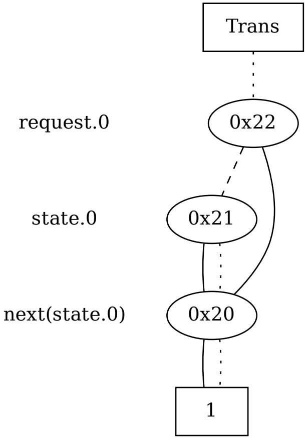
Automata for `short.soloready.smv`: I and R



OBDDs for `short.smv`: R

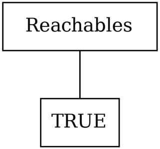


OBDDs for `short.soloready.smv`: R



OBDDs for `short.smv`: Reach

The one for soloready is the same



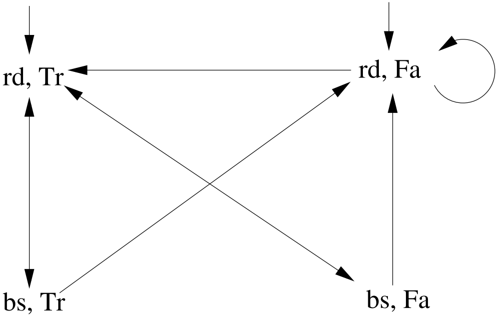
NuSMV Input Language

```
MODULE main
VAR
  request : {Tr, Fa};
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : ready;
                 esac;
SPEC
  AG((request = Tr) -> AF state = busy)
```

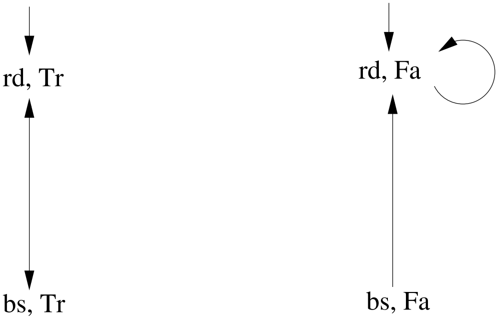
NuSMV Input Language

```
MODULE main
VAR
  request : {Tr, Fa};
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : ready;
                 esac;
  next(request) := request;
SPEC
  AG((request = Tr) -> AF state = busy)
```

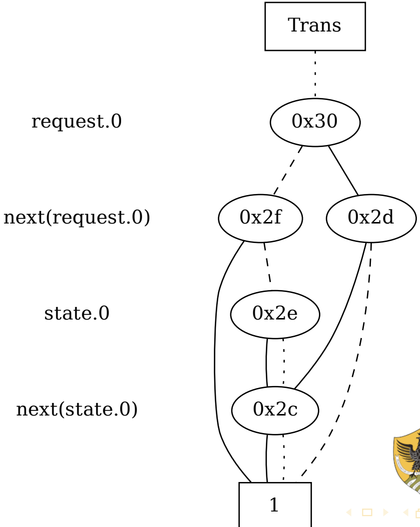
Automata for `short.soloready.smv`: I and R



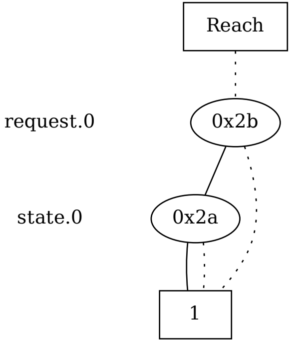
Automata for `short.soloready.req_const.smv`: I and R



OBDDs for `short.soloready.req_const.smv`: R



OBDDs for `short.soloready.req_const.smv`: Reach



OBDDs Pros and Cons

```
MODULE main
VAR
  m1 : 0..15; -- m1.0 is MSB!
  m2 : 0..15;
  m3 : 0..30;
ASSIGN
  next(m3) := m1 + m2;

SPEC
  AG(m3 <= 30);
```

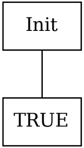
OBDDs Pros and Cons

```
MODULE main
VAR
  m1 : 0..15;
  m2 : 0..15;
  m3 : 0..30;
ASSIGN
  next(m3) := case
    m1*m2 <= 30: m1*m2;
    TRUE: m3;
  esac;

SPEC
  AG(m3 <= 30);
```

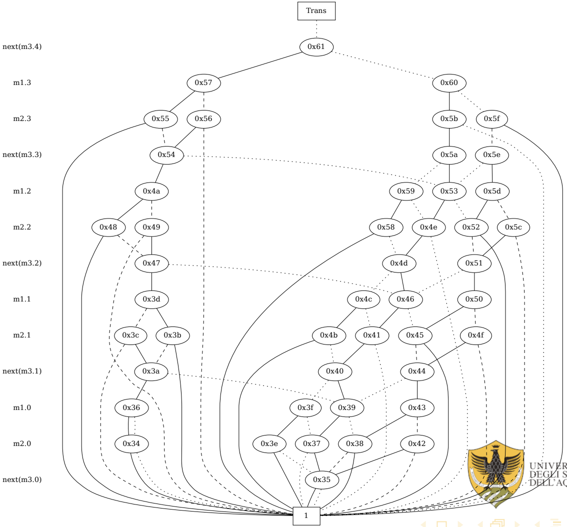
OBDDs for Adder and Multiplier: I

This is a set with $16 \cdot 16 \cdot 31 = 7936$ elements
Just one node to represent it...



UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

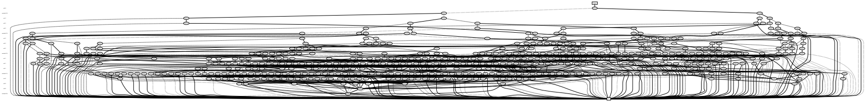
OBDDs for Adder: *R*



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

OBDDs for Multiplier: R



OBDDs Pros and Cons

- Number of variables is 13 for both models
  - 4 each for `m1` and `m2`, plus 5 for `m3`
- Number of BDD nodes:
  - adder: 47
  - multiplier: 538

OBDDs Pros and Cons

- No magic: SAT could be solved using OBDDs
  - just represent the instance with an OBDD and check if it is different from 0
  - very roughly speaking: if it were possible to solve it "efficiently" in this way, P=NP...
- Thus, there are boolean functions for which OBDDs representation is exponential, regardless of variable ordering
  - one example is the multiplier seen above
- It is not possible to say if OBDDs will be a good way to represent a problem, before trying it
  - for the adder, it is much more efficient
- Furthermore, finding a variable order in order to minimize the OBDD representation for a given function is an NP-complete problem

Computation of Least (Minimum) Fixpoint

```
OBDD lfp(MuFormula T) /* μZ.T(Z) */
{
  Q = λx.0;
  Q' = T(Q);
  /* T clearly says where Q must be replaced */
  /* e.g.: if μZ.λx.f(x) ∨ Z(x), then
     Q' = λx.f(x) ∨ Q(x) */
  while (Q ≠ Q') {
    Q = Q';
    Q' = T(Q);
  }
  return Q; /* or Q', they are the same... */
}
```

Computation of Greatest (Maximum) Fixpoint

```
OBDD gfp(NuFormula T) /* νZ.T(Z) */
{
  Q = λx.1;
  Q' = T(Q);
  while (Q ≠ Q') {
    Q = Q';
    Q' = T(Q);
  }
  return Q;
}
```

Symbolic Model Checking of **AG**p

- The idea is to compute the set of reachable states, and check if for all of them p holds
- $\mathrm{Reach} = \mu Z.\ \lambda x.\ [I(x) \vee \exists y :\ (Z(y) \wedge R(y, x))]$
    - of course, we get an OBDD on x as a result
    - recall that x (and y) is a vector of all boolean variables
- $\forall x \in S.\ \mathrm{Reach}(x) \rightarrow p(x)$
    - computationally easier: check that $\mathrm{Reach}(x) \wedge \neg p(x) = 0$
    - otherwise, we have a reachable state for which p does not hold...

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

```
bool checkCTL(KS 𝒮, CTL φ) {
   let 𝒮 = ⟨S, I, R, L⟩;
   B = LblSt(φ);
   return λx. I(x) ∧ ¬B(x) = λx. 0;
}
OBDD LblSt(CTL φ) { /* also 𝒮 = ⟨S, I, R, L⟩ */
  if (∃p ∈ AP. φ = p) return λx. p(x);
  else if (φ = ¬ϕ) return λx. ¬LblSt(ϕ)(x);
  else if (φ = ϕ1 ∧ ϕ2)
   return λx.LblSt(ϕ1)(x)∧LblSt(ϕ2)(x);
  else if (φ = EXϕ)
   return λx. ∃y : R(x, y)∧LblSt(ϕ)(y);
  else if (φ = EGϕ)
   return gfp(νZ. λx. LblSt(ϕ)(x) ∧ (∃y : R(x, y) ∧ Z(y)));
  else if (φ = ϕ1 EU ϕ2)
   return lfp(μZ. λx. LblSt(ϕ2)(x)∨
     (LblSt(ϕ1)(x) ∧ (∃y : R(x, y) ∧ Z(y))));
}
```

Towards Bounded Model Checking

- Explicit and symbolic model checking are good, but many systems cannot be checked by neither
  - RAM and/or execution time are over soon
- Symbolic model checking directly makes use of boolean formulas through OBDDs
- What about using CNF, so that SAT solvers can be employed?
  - modern SAT solvers are pretty good in many practical instances
  - notwithstanding the SAT problem is of course still NP-complete

Towards Bounded Model Checking

- One big problem: computing quantization, AND, OR and negation of a CNF is not straightforward
  - especially because instances from Model Checking are HUGE
  - also checking equivalence of two CNF is not trivial, as CNF is not canonical
- However, if we set a limit k to the length of paths (counterexamples), then most of this is not needed any more
  - copy R for k times, with small adjustments
- This is actually *bug hunting*: if the result is PASS, then there is not an error within k steps
  - but there could be one at $k+1$...
  - however, this is better than simple testing, as errors within k steps can be ruled out

Bounded Model Checking of Safety Properties

- In Bounded Model Checking (BMC) we are given a KS $\mathcal{S} = \langle S, I, R, L \rangle$, an LTL formula φ, and $k \in \mathbb{N}$ (also called *horizon*)
- Let us consider the LTL property $\varphi = \mathbf{G}p$, being $p \in AP$
- We want to find counterexamples (if any) of length exactly k
- If $x = x_1, \ldots, x_n$ with $n = \lceil \log_2 |S| \rceil$, let us consider $x^{(0)}, \ldots, x^{(k)}$
- $\mathcal{S} \models_k \mathbf{G}p$ iff the following CNF is unsatisfiable:

$$I(x^{(0)}) \wedge \bigwedge_{i=0}^{k-1} R(x^{(i)}, x^{(i+1)}) \wedge \neg p(x^{(k)})$$

  - otherwise, a satisfying assignment is a counterexample

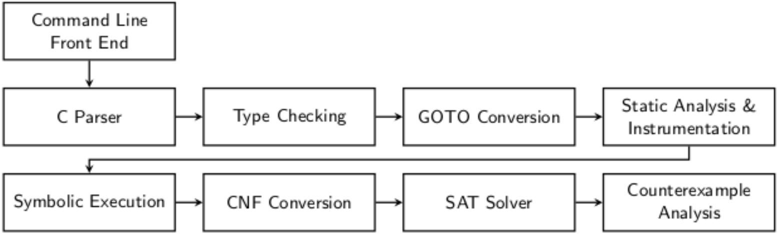
Bounded Model Checking of Safety Properties

- Note that each $x^{(i)}$ encloses n boolean variables, thus we have $n(k+1)$ boolean variables in our SAT instance
  - the longest our horizon, the biggest our SAT instance
- Note that I and R must be in CNF, which is not difficult
  - NuSMV does this pretty well
- It is straightforward to modify the previous formula to detect counterexamples of length *at most* k
- However, it is usually preferred to perform BMC with increasing values for k
  - practically, till when the SAT solver goes out of computational resources
  - some approaches exist to estimate the *diameter* of a KS...

Bounded Model Checking of Programs

- Till now, we had to write a model of the system under verification (SUV)
- There are some cases in which we can use the actual SUV, with little or no instrumentation
  - it is possible to translate a digital circuit to a NuSMV specification in a completely automated way (not difficult to imagine how...)
  - here, we want to deal with a rather surprising application of BMC: model checking a C program!
- CBMC is a model checker performing BMC of C programs with little or no instrumentation
  - thus, the input for CBMC is a C program (possibly with some added statements)
  - an integer k may be required too
  - again, output is PASS or FAIL (with a counterexample)
- We now give the main ideas of how it works

CBMC


Command Line
Front End
C Parser
Type Checking
GOTO Conversion
Static Analysis &
Instrumentation
Symbolic Execution
CNF Conversion
SAT Solver
Counterexample
Analysis