Telematics
Computer Systems

Freie Universität Berlin

Model Checking

Marcel Kyas

Freie Universität Berlin

Winter 09/10

Organisation

Course outline

Model checking

Lecturer: Marcel Kyas

2 hrs lecture + 2 hrs tutorial

8 ECTS (5 lecture + tutorial, 3 mini project)

Visit me during my office hours if You have questions, comments, problems, . . .

Office hours: Thursday 14–15, Room 106 and upon request

e-mail: `marcel.kyas@fu-berlin.de`

Register and stay informed

- For the course at `https://www.mi.fu-berlin.de/kvv/course.htm?sid=18&cid=8298&iid=1`
- For the mailing list at `https://lists.spline.inf.fu-berlin.de/mailman/listinfo/model-checking-2009`
- Check the course page at `http://cst.mi.fu-berlin.de/teaching/WS0910/19612-V-Modelchecking/index.html`

All announcements (e.g., change of date or room) will be sent on the mailing list!

Course language

- All course material are in *English*.
- This lecture will be held in English, too.
- Upon request, the tutorial may be held in German, if nobody objects.
- You are allowed to answer the assignments, exam, and questions in either English or German.

Lectures

- *Lectures* are (usually) Fridays (12–14) in SR006. Watch the course's web page for changes.
  - October 14, 16, 23, 30
  - November 13, 20, 27
  - December 3, 11(?), 18
  - January 8, 15, 22, 29
  - February 5
- Observe that this fixes 14 of 15 lectures (16th lecture is exam). We need to find at least another date.

Tutorials I

- *Tutorials* are (usually) Wednesdays 18:00–19:30 (if no one objects) in SR006. Again, watch the course's web pages for changes.
  - October 21, 28,
  - November 11, 18, 25
  - December 1, 9(?), 16
  - January 6, 13, 20, 27
  - February 3, 10
- Observe that this fixes 13 of 15 tutorial sessions. We need to find another date.
- Tutorials will consist of theoretical exercises and practical work.

Tutorials II

- To qualify for the exam you have to participate in all but two tutorials.
- You have to submit all but two assignments.
- Each assignment will be graded. You need to score at least 60% of the possible score.
- The assignments are designed for team-work. Submit each assignment in groups of two. Do not work alone and do not split the work. Collaborate!

Written exam

Written exam on Friday, February 12, 12–14

- You may take a *hand-written* summary of 4 pages (2 sheets) to the exam. You are kindly requested to submit your sheets together with the exam.
- This is the *only* material admitted.

Mini-project

- A final part of this courses examination is a *mini project*.
- It should take You 2–3 weeks to finish the project (3 ECTS correspond to 90 working hours!)
- You are supposed to apply the methods You learn in this lecture and tutorials to model Your own system, formulate your own requirements, and use model checking to verify the model.
- You *have to* deliver the models and requirements
- You *have to* hand in a *written report* describing your model, the requirements, and how you developed it. The report should have 10–15 pages.
- We fix the topic on January 6. You must hand in your report and the model no later that February 26 (hard deadline).
- When you deliver late, your report is marked as fail!

Final Grade

- The grade for this course is obtained from the grade of the exam and the grade of the mini-project.
- If you fail either, you fail the whole course.
- If you pass the exam and the mini-project (both grades $\leq$ 4.0, your final grade will be the average of both.
- Example: Exam 5 and mini project 2 results in a final grade of 5
- Example: Exam 3 and mini project 1 results in a final grade of 2.
- If you fail the exam, you can take it once again.
- If you fail the mini-project, you can receive a new version after two weeks.

Lecture notes

- There are no lecture notes yet. I hope to have them ready until mid-term (before Christmas)
- Try to find and use the books and take notes
- I may put the slides on the web-page (accessible only within the university network)

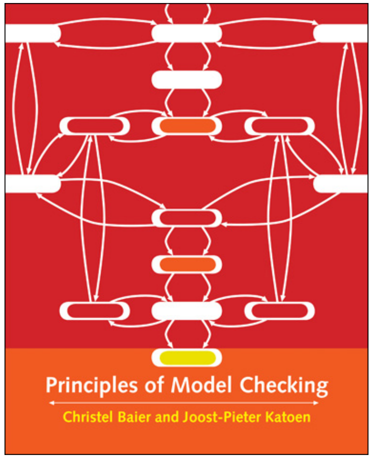
Literature

Christel Baier and Joost Pieter Katoen

Principles of Model Checking

MIT Press

2008



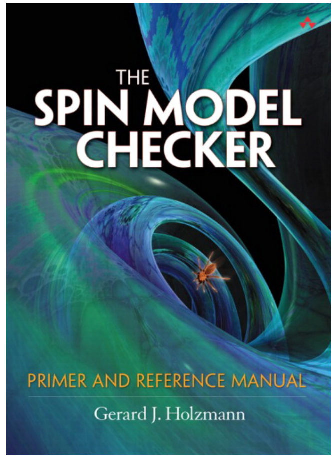
Literature

Gerard J. Holzmann

The SPIN Model Checker

Addison Wesley

2004



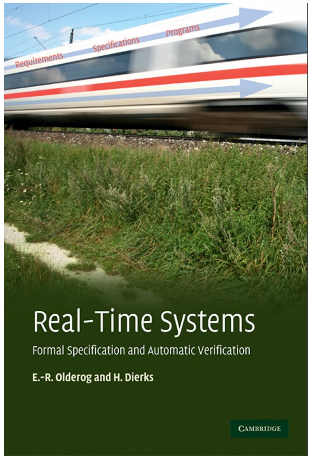
Literature

Ernst-Rüdiger Olderog and Henning Dierks

Real-Time Systems: Formal Specification and Automatic Verification

Cambridge University Press

2008



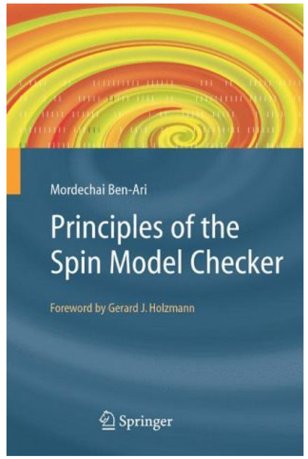
Literature

Mordechai Ben Ari

Principles of the SPIN Model Checker

Springer-Verlag

2008



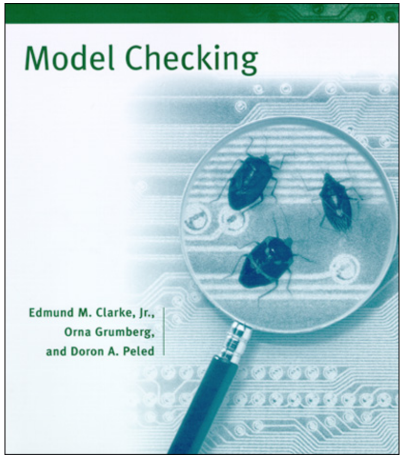
Literature

Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled

Model Checking

The MIT Press

1999



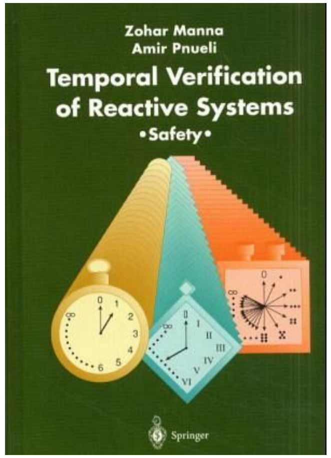
Literature

Zohar Manna and Amir Pnueli

Temporal Verification of Reactive Systems: Safety

Springer Verlag

1995



Introduction

Simplicity is prerequisite for reliability.

Introduction

- What is the purpose of model checking?
- Is model checking relevant?
- Is model checking feasible?
- How does it work?

The cost of fixing a bug

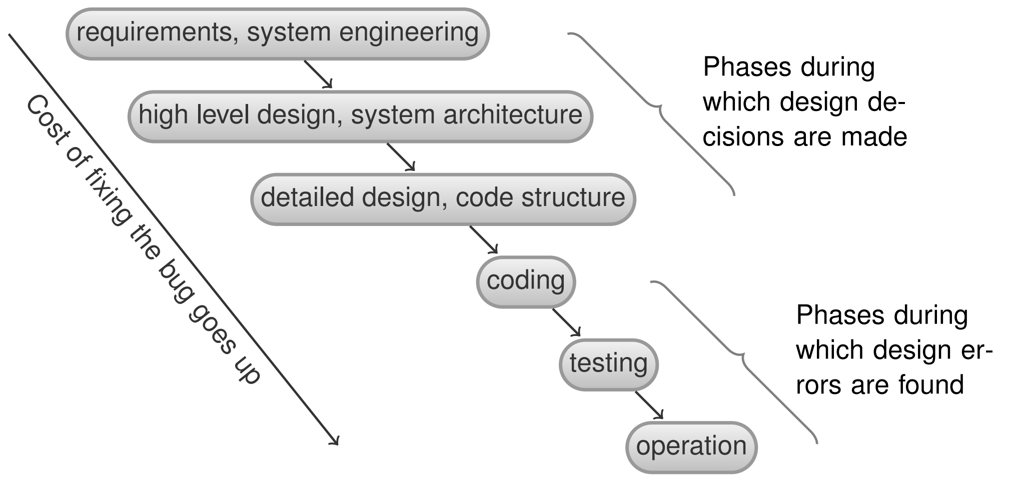


Figure: The cost of fixing a bug

The cost of fixing a bug

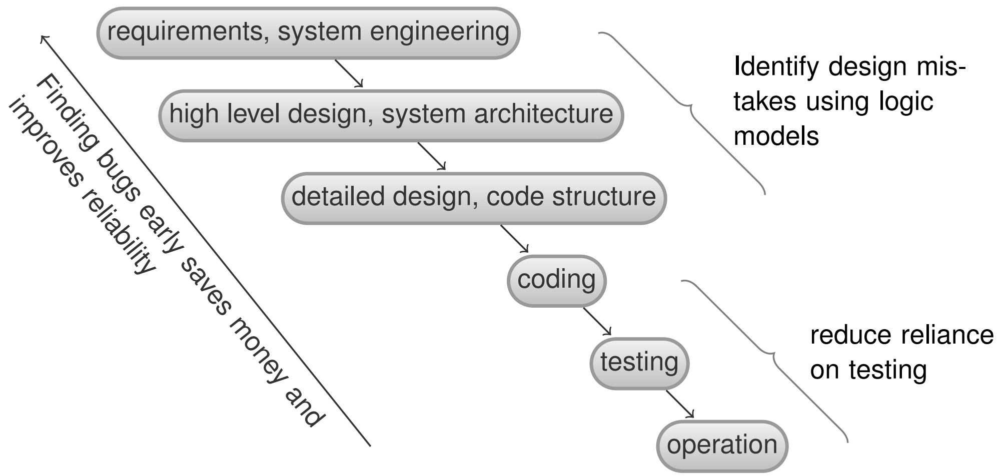


Figure: Reducing the cost of fixing a bug

Why is model checking interesting?

- Model checking is a mature engineering method
  - distinguishes between requirements and specifications
  - uses engineering prototypes to analyse designs
  - can predict the essential properties of a design before construction begins
- basic engineering approach to the design of concurrent software designs could look as follows

requirements	$\longrightarrow$	logic
prototype	$\longrightarrow$	logic model
analysis	$\longrightarrow$	model checking

What do software developers do?

- There is not much engineering in software engineering today
- Instead of modelling and analysing
  - reuse: copy something that seemed to work before
  - trial and error testing
- How does this approach work today?
  - Testing takes more than 50% of the development effort
  - major errors still slip through
  - classic testing methods were never meant to be applied to concurrent software

Testing shows the presence, not the absence of bugs

Edsger W. Dijkstra

Number of executions

Lemma

Let n be the number of threads, $\langle p_i \rangle_{i<n}$ a sequence of the number of atomic instructions of each thread, starting with 0. Then the number of executions is:

$$\frac{\left(\sum_{i=0}^{n-1} p_i\right)!}{\prod_{i=0}^{n-1} p_i!}$$

Small programs already show many executions

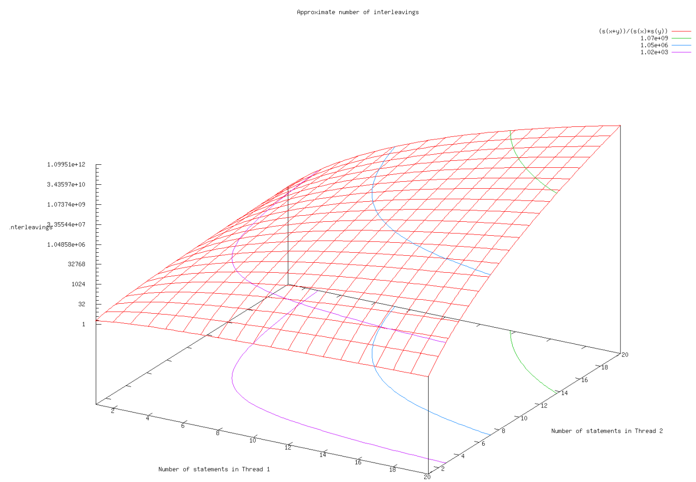
Example

If all statements are atomic, then the example program has $\langle 3,2 \rangle$ statements per thread, which results in $\frac{5!}{3! \cdot 2!} = 10$ different executions.

Example

The byte code of the same example, it has $\langle 13,9 \rangle$ instructions per thread, which results in $\frac{(13+9)!}{13! \cdot 9!} = 497420$ different executions.

More examples



Consequences I

- Real systems have much more executions, 10^{90} different ones are often seen.
- Testing is infeasible:
  - The number of test cases gets really large, usually one per execution.
  - Assuming 10 tests per seconds, it will take over $400,000$ years to test all executions of two threads with 25 instructions each.
  - We can hardly the execution that is actually chosen when executing the test.

Consequences II

Since testing is impossible in practice, other methods like *program verification* have to be used.

- Inductive methods like *(bounded) model checking* try to build suitable representations of all possible executions, e.g. transition diagrams, and check all possible executions
- Deductive methods use some kind of proof technique, e.g. Hoare-style calculi, to prove properties of the program.

The software crisis

- A *software crisis* was first declared in 1968
  - complexity of software grew faster than our ability to control it. In 1968, a “large” program was about 100,000 lines of code.
  - the crisis still persists
- today
  - Large programs became 10,000 times larger
  - avalubale memory became 100,000 times larger
  - computers became 1,000,000 times faster
  - multi-processing became ubiquitious
- but software is basically still developed and tested the same way as 40 years ago
- we still cannot predictably produce reliable software
- solving this problem is the single most-important challange in computing science today

The cost of buggy software

- Software bugs are catastrophic for the development process
  - fewer than 1 in 3 industrial software projects proceed as blanned
  - more than 1 in 5 projects fail completely
  - Inadequate testing of software costs 59.5 billion $ each year (RTI, 2002)

Software bugs can kill people

- The “Therac-25 incident” (1985–1987)
- China Airlines B1816, Airbus A300 crash (Nagoya, 1994)
- The “Ariane 5 Flight 501” (1996)
- Therac-25 deja-vu (2009)

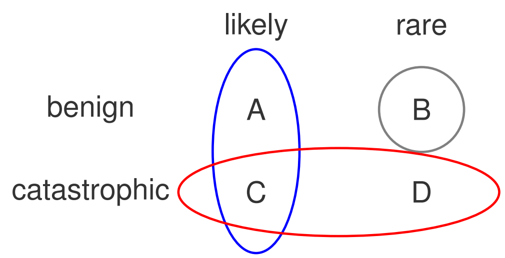
Software bugs can cause great economic damage

- Wall-street crash (1987)
- Pentium processor division algorithm (1994)
- Denver Airport: computerised baggage handling fails (1995)

Software bugs can affect us

- AT&T phone system crash (1986)
- Rounding errors in election software (e.g. Schleswig-Holstein, 1992)
- Thai minister trapped in BMW by central locking system (2003)

What exactly does model checking address?



- likely bugs may be addressed with testing
- bugs with catastrophic results are addressed by model checking
- model checking helps with *rare* bugs (Heisenbugs)

Moore made it feasible!

- Improved algorithms reduced the memory requirements and run-time
- But machines got faster and have more available memory
- A problem that needed 7 days in the 1980s can be solved in a couple of seconds today

Model checking in a nutshell

$$\mathscr{M} \overset{?}{\models} \varphi$$

- Given a model $\mathscr{M}$, decide whether it satisfies property φ

A reformulation

- Given
  - A system S with system behaviour $L(S)$ (the model)
  - A property φ describing valid/desirable behaviours $L(\varphi)$
- Prove $L(S) \subseteq L(\varphi)$
  - To prove this, we can prove $L(S) \cap (\Sigma^\omega \setminus L(\varphi) = \emptyset$
  - Which is the same as

$$L(S) \cap L(\neg\varphi) = \emptyset$$

Refutation principle

- When $L(S) \cap L(\neg\varphi)$ is empty, then S satisfies φ
- When $L(S) \cap L(\neg\varphi)$ is not empty, then it contains a counter example proving that S violates φ
- The value of model checking is the *counter example*

Scope

Logic model checking can catch a range of logic and functional design errors, especially errors related to concurrency and multi-threading.

- deadlock, livelock, starvation
- race condition
- locking problems, priority problems
- resource allocation errors
- reliance on relative speeds of thread execution
- violations of known system bounds
- specification incompleteness
- specification redundancy (dead code)
- logic problems: missing causal or temporal relations

What do we need to do?

To use model checking successfully, we have to:

- build logic models
- how to express requirements
- how to analyse models and requirements

These are basic engineering skills!

A short timeline

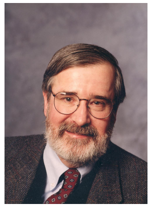
1936	first theory on computation (A. Turing)
1955	early work on tense logic
1960	theory of ω-regular languages (J.R. Büchi)
1968	the term “software crisis” is introduced
1975	Guarded Command Language (E.W. Dijkstra)

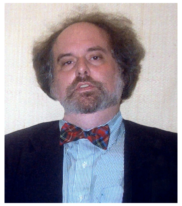
A short timeline

1976–1979	first attempts at reachability analysers
1977	linear temporal logic for system verification (A. Pnueli)
1978	communicating sequential processes (C.A.R. Hoare)
1980	earliest predecessor of SPIN (“pan”)

A short timeline

1981 E. Clarke and A. Emerson introduce the term “model checking” and the logic CTL
1981 J. Sifakis and J.P. Queille develop a model checker
1986 Automata-theoretic framework of LTL model checking (P. Wolper and M. Vardi)





A short timeline

1986 Reduced, Ordered, Binary Decision Diagrams (R. Bryant)
1986 Mazurkiewicz’ trace theory
1989 SPIN version 0
1993 SMV (K. McMillan)
1995 Partial order reduction and LTL conversion (D. Peled)
2008 E. Clarke, A. Emerson, and J. Sifakis receive the ACM Turing Award for model checking

Success stories I

- Mars pathfinder (1997 mission)
- two typical software requirements:
  - mutual exclusion
  - potential priority inversion

Success stories II

- AT&T phone system crash
  - A mis-placed break statement brought the AT&T phone system down for 9 hours (snow-ball effect)
  - cost (w/o collaterals: $75,000,000 in industry and $100,000,000 at AT&T’s customers
- two typical software requirements:
  - mutual exclusion
  - potential priority inversion

Course overview

1. The distinction between programming and modelling.
2. Using the model checker SPIN and its input language PROMELA.
3. The automata theoretic foundation of SPIN and its algorithm.
4. Specifying properties by observers and in temporal logics.
5. Automatic state-space reduction techniques.
6. Symbolic model checking using binary decision diagrams.
7. Real-time systems and timed automata
8. The model checker UPPAAL
9. Comparing models and understanding the properties they share.

Modelling vs. programming

Models in computing

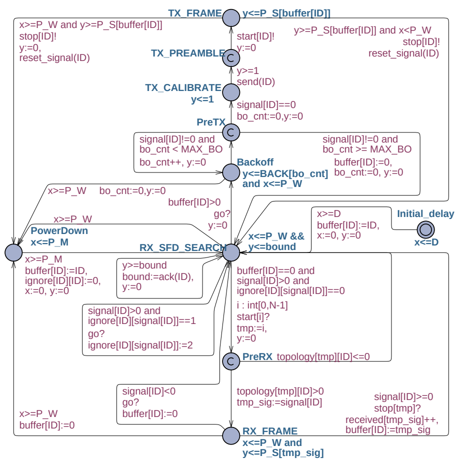


Figure: A model in computing!

What is modelling?

Modelling is a universal activity by which we form “abstractions” of our environment.

Example

Small children learn to build models. A baby sees a pretty red disc on top of a stove and touches it. She learns that cherry red objects are painful. Later, she sees a read coal that falls out of a fire place. She may *apply* her model now and save herself from burning her hands. She may also *validate* the model. Later, when her parents go to a party, she sees her mother wearing a red dress. The mother is baffled at her child screams of panic—the model, valid or not, does not apply.

What is a model?

- A *model* is a simplified representation of some real-world entities or phenomena.
- They are (usually) obtained from observing the environment or analysing planned systems
- We model to better understand it:
  - Focus on interesting aspects
  - Predict and visualise potential outcomes
  - Create mechanisms to test and verify and approach *before* implementing it

What is a model?

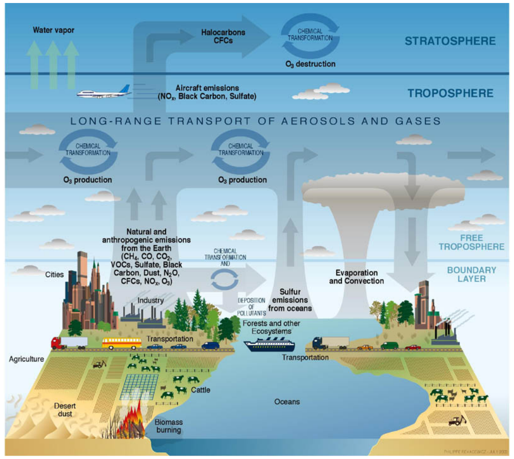


Figure: Model of atmospheric processes

What do we model in computing science?

- In computing science, a model is a *mathematical object* that describes systems, i.e. structures and processes
  - Interactions between actors and programs (electronic purse)
  - Chemical processes (batch processing plant)
  - Physical processes (thermostat, air bag)
  - Complex systems (autopilot, telephony networks, electrical networks)
- Thus modelling is (unfortunately) closely related to programming
- Models should also be understood as *simplified* representations of programs
- It is not necessary to model programs (except when we are supposed to re-engineer it or understand it), since each program is already a model of a process.
- Usually, programs should be the result of models and not the opposite way around.

Modelling vs. programming

A *program* is a formal description of a system that is used by a computer to perform operations.

Programs are models

The main difficulty in distinguishing “programs” from “models” is, that any “program” is a model of a process that is performed by a computer. But a program usually omits central parts of an *underlying* model and makes assumptions that are implicit.

Modelling vs. programming

- A model of a program or system needs not be *executable,* but all programs are
- A model needs not be *deterministic,* but programs hopefully are.
- A model needs not be *complete,* but programs have to (otherwise they crash of resource exhaustion or do not do anything useful)

Characterisation of a model

- Models generalise program in that they *abstract* many details. A model will not bother with user interfaces and memory management, unless these are the *aspect* to model.
- Models serve a *purpose* different from programs. A program is supposed to execute a specific task while a model is used to make predictions about a systems behaviour.
- Models must eventually be related to a system and the programs of the system, e.g. by serving as specifications or *test oracles.*
- Finally, a model can be formally checked against a program. This is seldom done, because it is computationally hard or infeasible, which is a task similar to proving correctness using Hoare's logic.

Models and programs are communication!

- In any case, models and programs must also be understood as communication
- Thus they must convey information that must be understood by others.
- A model or program not understood by a colleague is rubbish!
- A personal opinion (drawn from a lot of experience): UML is *not* suited as a modelling language, because it lacks a precise, unambiguous semantics. Anything that enables two persons forming different opinions on what a model means is unsuitable!

Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer.

E.W. Dijkstra

Occam's razor

entia non sunt multiplicanda praeter necessitatem

Attributed to William Ockham (c. 1285–1349)

entities must not be multiplied beyond necessity

Synchronisation skeletons I

- A classic example of applying Occam's razor in our context are *synchronisation skeletons*
- A synchronisation skeleton expresses all necessary information about a *mutual exclusion algorithm*
- In fact, these algorithms are usually expressed by synchronisation skeletons
- A synchronisation skeleton omits all code that is not necessary for understanding how synchronisation works.

Synchronisation skeletons II

```
active proctype P1() {
     Remainder:
       skip;
     Entry:
       atomic { b[0] = true; turn = 0; }
       atomic { (!b[1] || (turn == 1)) -> in_cs++; }
     Critical:
       assert (in_cs <= 1);
     Exit:
       atomic { in_cs--; b[0] = false; }
       goto Remainder;
}
```

Figure: Peterson's algorithm. The syntax is explained later

Synchronisation skeletons III

- We do not consider what the algorithm does in Remainder or the critical section
- We make assumptions (that are left implicit in the previous slide):
  - Reading from a variable and writing two a variable are indivisible steps (Peterson uses the *weakest* atomicity assumption: only one bit is read or written)
  - The code in remainder does not write to `turn`, `b[0]`, `b[1]`, and `in_cs`
  - The code in the critical section does not write to `turn`, `b[0]`, `b[1]`, and `in_cs`
- As long as we adhere to these assumptions, any *use* of that skeleton provides the desired property: the critical sections are executed in mutual exclusion.

Understanding the difference

- The crucial differences between *models* and *programs* are, that most models contain aspects of a system that are not implementable in computers

Example: Mode confusion

- Complex user interfaces, like ones used in air planes, are multi-modal
- Availability of controls depends on the planes mode (you cannot extend the landing gear when over a certain height or flying faster than a specific speed)
- Mode confusion means that the pilot assumes that the system is in a specific mode when it isn't
- It arises from a discrepancy between the mental model of the plane's controls and the actual model

Using model checking to identify mode confusion

- Build a model of the system
- Build a model of the pilot's mind
- Check whether there is a discrepancy
- Note: The pilot's mind is not executed on a computer

Correctness of programs

Clearly every program does something and therefore every program is correct if we are willing to accept whatever it does. Generally, however, the program was written for some purpose, and so we have some prior idea of what we want it to do. If this idea can be made precise, then it is called the specification of the program. A program is correct only relative to a specification, that is, it is correct if what it does is what the specification demands that it do. (Parikh, 2001)

A system cannot be correct unless its correctness depends only upon events and conditions observable within the system. (Lamport, 1976)

Wovon man nicht sprechen kann, darüber muss man schweigen. (Wittgenstein, 1922)

Three ingredients

Requirements

The properties an entity should satisfy

System

The entity we want to construct.

Assumptions

The "environment" we want the system to work in and the platform it is going to execute. More precisely: The *assumptions* of the system's operational context.

Correctness revisited

A *reactive system* is correct when it satisfies its requirement in every environment that satisfies our assumptions.

SPIN and Promela

What is Promela?

- Promela is an abbreviation for *process meta-language*.
- It is the input language of the model checker SPIN.
- Today, we describe the complete language.
- Promela's ancestors are *Dijkstra's guarded command language* and *Hoare's communicating sequential processes (CSP)*

SPIN

- Now we treat SPIN as a *black box*
- SPIN includes a *simulator* that executes Promela models.
- SPIN also includes a *explicit state on-the-fly model checker*
  - *explicit state* means: a complete state graph is built (nodes represent states and are labelled with valuations of variables, edges connect nodes that are related by a state transformation)
  - *on-the-fly* means: the state graph is constructed when needed.
  - *model checker* means: it searches the state graph (the *semantic* model) for property violations.
  - You do not need to understand all this, because we will look at all this in the next lecture.

A word of caution

- The following description contains some simplifications and white lies.
- We will elaborate these issues during the tutorial session.
- As with any formal language, Promela and Spin can sometimes be surprising, because it behaves in unexpected but completely logical ways.

Promela

- We introduce Promela by means of a series of examples of increasing complexity
- We also show how Promela models can be simulated and analysed using SPIN
- We advise you to try to run and analyse all models, so go and install SPIN after this lecture
  - SPIN is distributed at `http://spinroot.com`
  - SPIN is available on our machines.
  - Under Linux, add the directory `/import/Spin/bin` to your `PATH` variable, e.g., by adding `export PATH=$PATH:/import/Spin/bin` to your `$HOME/.bashrc`

Hello, world

```
active proctype main()
{
  printf("Hello,␣world\n")
}
```

Simulating this with SPIN results in:

```
$ spin hello.pml
Hello, world
1 process created
```


Process types

- **`proctype`** is a keyword in spin
- It defines a type of *processes*
- Processes have a name (e.g., `main`) and they have formal parameters (e.g., `()` for no parameters or `(int id)` for a single parameter called `id` of type **`int`**)
- A process then declares a block (`{ ... }`) containing (a sequence of) statements and variable declarations.
- **`active`** is a second keyword to modify a **`proctype`**. A process of that type will be started at initialisation time.

init process

- If the type and name of the initial process is not important, we can use the keyword **init** instead.
- **init** cannot be used for any other purpose.

```
init {
  printf("Hello,␣world\n")
}
```


Statements

- **printf**(...) is a statement that behaves just like the corresponding C statement.
- Note that there is no ; in our example.
  - ; is a statement *separator*
  - Promela allows redundant ; and will ignore it

Peterson's mutual exclusion algorithm I

We use *Peterson's mutual exclusion algorithm* as our running example.

```
bool b[2];
byte turn;
byte in_cs;
```

- Variables in Promela are typed and must be declared.
- A variable is *global* and *shared* when declared outside of any function or process.

Types

Type	Typical range
bit	0, 1
bool	**false**, **true**
byte	0, ..., 255
chan	0, ..., 255
mtype	0, ..., 255
pid	0, ..., 255
short	$-2^{15}, \ldots, 2^{15}-1$
int	$-2^{31}, \ldots, 2^{31}-1$
unsigned	$0, \ldots, 2^n - 1$ with $1 \leq n \leq 32$

Table: Types in Promela

Arrays

- Only *one-dimensional* arrays are supported.
- Arrays are declared like in C: **`byte`** `a[12]`
- Arrays can be initialised at declaration time **`int`** `a = 42;` and **`int`** `b[3] = 1`.
- Array initialisation initialises all members.
- Array indices range from 0 to $N-1$, where N is the length of the array.
- Default initialisation is 0.

Peterson’s mutual exclusion algorithm II

```
active proctype P1() {
    Remainder:
      skip;
    Entry:
      atomic { b[0] = true; turn = 0; }
      atomic { (!b[1] || (turn == 1)) -> in_cs++; }
    Critical:
      assert (in_cs <= 1);
    Exit:
      atomic { in_cs--; b[0] = false; }
      goto Remainder;
}
```


Peterson’s mutual exclusion algorithm III

```
active proctype P2() {
    Remainder:
      skip;
    Entry:
      atomic { b[1] = true; turn = 1; }
      atomic { (!b[0] || (turn == 0)) -> in_cs++; }
    Critical:
      assert (in_cs <= 1);
    Exit:
      atomic { in_cs--; b[1] = false; }
      goto Remainder;
}
```


Expressions

- Expressions are built like C or Java expressions, and follow roughly the same precedence rules.
- Unlike C, all Promela expressions are free of side-effects.
- One exception: **`run`** `main()`, where `main` is some **`proctype`** creates a new process and returns its PID.
- A value of 0 means that the maximum number of processes (255) has been created.

Statements

- **skip** is a statement that does nothing (except for allowing more interleaving)
- `b[0] = true` is a classical assignment.
- Statements are *composed* by **;**
- Labels are variables followed by **:**, e.g. `Remainder:`. They can be put in front of any statement.
- **goto** `Remainder` resumes execution at `Remainder:`. Some labels (e.g. when they start with `accept`) have a special meaning.
- **assert** `(in_cs <= 1)` is an assertion. The *simulator* indicates the assertion violation and aborts execution while the *verifier* checks it like an *invariant* (precise definition pending)

Guarded commands

- Look at the *guarded command* `(!b[0] || (turn == 0)) -> in_cs++;`
- Technically, the `->` means the same as **;** but provides a better *visual* clue of the intended meaning.
- `!b[0] || (turn == 0)` is a Boolean expression.
- Every boolean expression is a statement.
- Execution proceeds when the condition is **true** and block while it is **false** (like **await**)
- The combination `(!b[0] || (turn == 0)) -> in_cs++;` means: "wait until `!b[0] || (turn == 0)` holds, then execute `in_cs++;`"
- Understand that there is a scheduling point between the test and the reaction!
- As with C, every expression may be treated as a Boolean expression, where 0 is **false** and anything else is **true**.

atomic statements

- **atomic ...** declares that the execution of statements inside the braces is atomic, i.e. no interleaving is possible.
- Statements inside of **atomic** may be non-deterministic
- If a statement is not *enabled* (a boolean condition is false), the statement is interrupted and another process may execute. Then, the process may be reelected nondeterministically, and execution resumes as if nothing has happened.
- Style: Avoid interruptible atomic statements! Make sure that the execution inside an atomic statement always terminates

d_step statements

- **d_step ...** declares that the execution of statements inside the braces is atomic and deterministic, i.e. no interleaving is possible.
- Conditions:
  - The execution inside must be deterministic. If there is non-determinism, always the same behaviour will be chosen. But: which one is undefined
  - No **goto** jumps into or out-of **d_step**s. Warning: `L:` **d_step skip** jumps *into* the **d_step**, so write `L:` **skip; d_step skip**
  - The execution of a **d_step** may not be interrupted by blocking. It is an error if execution is blocking inside.
- `d_steps` can be used for defining our own statements (Later).

Conditional statement

```
if
:: (b1) -> s1
:: (b2) -> s2
:: (b3) -> s3
:: else s4
fi
```

- Works like a case/if statement: choose a branch whose condition is true and if none is true, continue with **else**.
- BUT: If two conditions are true, any branch may be chosen (non-determinism)
- Need not start with a condition, could also be a statement (then the condition is **true**).

Loop statement

```
do
:: (b1) -> s1
:: (b2) -> s2
:: (b3) -> s3
:: else s4
od
```

- Works like an infinite loop: choose a branch whose condition is true and if none is true, continue with **else**. And then repeat!
- Like **if**: If two conditions are true, any branch may be chosen (non-determinism)
- End loop with **break** or **goto**

Expressing the same assignment

```
b = 1;

if
:: b = 1;
fi

if
:: true -> b = 1
fi

do
:: true -> b = 1; break
od
```


Comment: way's to encode busy waiting I

```
(!b[0] || (turn == 0)) -> in_cs++

if
:: (!b[0] || (turn == 0)) -> in_cs++
fi

Loop: if
  :: (!b[0] || (turn == 0)) -> in_cs++
  :: else goto Loop
  fi
```

Comment: way's to encode busy waiting II

```
do
:: (!b[0] || (turn == 0)) -> in_cs++; break
:: else skip
od

do
:: (!b[0] || (turn == 0)) -> in_cs++; break
:: else
od

do
:: (!b[0] || (turn == 0)) -> in_cs++; break
od
```


Understanding the differences

- All these examples have slight semantic differences which may be completely irrelevant to your problem.
- The question is: How many interleavings do I want to allow? How close do I want to model the real system!
- For **`if`**-statements: For a classical **`if`**, add an **`else`** and make sure that all possibilities are disjoint.
- Similar for **`do`** loops

Symbolic constants

- **`mtype`** is an enumeration type.
- Now constants are defined by the declaration `mtype={ ... };`.
- Multiple declarations will merge the definitions and will not redefine definitions: `mtype={P,C};mtype={N};` is the same as `mtype={P,C,N};`.
- No more than 255 symbolic names can be declared in *all* mtype declarations combined.

Data structures

- Promela supports simple data structures

```
typedef Point {
  int x = 1, y
};
```

- New records of type `Point` will have field `x` initialised to 1.
- Structures may nest.
- Structures are *not* recursive.
- Using structures you can create arrays of arrays.

```
typedef Array { byte el[4] }; Array a[4];
a[1].el[2] = 1
```

A note on memory management

- Promela does not have pointers,
- no malloc() or free()
- everything is *statically allocated* for each process
- Hint: Don't use processes to mimic data structures
- Remember to model and not to program

inline definitions

```
inline P(s) {
  d_step { (s > 0) -> s--; }
}

inline V(s) {
  d_step { s++; }
}
```


SPIN uses call-by-name

- When calling `P(s)`, SPIN inserts the **`inline`** body and substitutes the formal parameter by the actual one.
- Side effects are visible outside of the **`inline`** body.
- But we get nicer error messages.

Simple semaphore example

```
proctype producer() {
  byte data;

  do
  :: P(emp);
     data++;
     buf = data;
     printf("produced␣%d\n", data);
     V(ful)
  od
}
```

Channels

- We have seen the first part of Promela that already is adequate to deal with shared variable concurrency.
- Many problems are inconvenient to model using shared variables.
- Promela also provides channel based communication

Ping-pong

```
mtype = { ping, pong }

chan M = [1] of { mtype };
chan W = [1] of { mtype };

active proctype P1()
{
    do
    :: M!ping;
       W?pong
    od
}
```


Channels

- Channels are variables of type **`chan`**
- `[1]` designates the channel capacity (1 message)
- `{ mtype }` declares the type of messages
- Channels can carry many values: `{ byte, byte, bool }` sends triplets of messages.
- Triplets are written as comma separated lists.
- When you define message types with **`mtype`** as first component, you might also write something of the form `M(x,y,z)` instead of `M,x,y,z`, when `M` is a constant.

Sending and receiving a message

- The statement `M!ping` sends the message `ping` on the channel `M`
- When the channel expects to send more values, we separate them by comma: `C!1,x*y,false`

Receiving a message

- The statement `M?ping` receives the message `ping` on the channel `M`
- It waits until the oldest message in the queue is a `ping` message.
- If `v` is a variable, then *M?v* receives the oldest message and assigns the value to `v`.

An extended example I

```
active proctype Mproc()
{
    W!ini;                  /* connection      */
    M?ack;                  /* handshake       */
    timeout ->              /* wait            */
    if                      /* two options:    */
    :: W!shutup             /* start shutdown  */
    :: W!dreq;              /* or request data */
        do
        :: M?data -> W!data            /* send data       */
        :: M?data -> W!shutup; break   /* or shutdown     */
        :: timeout -> W!shutup; break
        od
    fi;
    M?shutup;               /* shutdown handshake */
    W!quiet;
    M?dead
```


An extended example II

```
active proctype Wproc()
{
    W?ini;                  /* wait for ini    */
    M!ack;                  /* acknowledge     */
    do                      /* 3 options:      */
    :: W?dreq ->            /* data requested  */
        M!data              /* send data       */
    :: W?data ->            /* receive data    */
        skip                /* no response     */
    :: W?shutup ->
        M!shutup;           /* start shutdown  */
        break
    od;
    W?quiet;
    M!dead
```


Question: Does this protocol contain a deadlock?

- Lets use SPIN to analyse the model:
  1. `spin -u1000 -c datatrans.pml` simulates the protocol for up to 1000 steps and shows a message sequence chart (MSC). Executing this a couple of times showed that the protocol works as intended.
  2. `spin -a datatrans.pml` generates a *verifier*, which is a C program that contains a model checker and the model described in `datatrans.pml`
  3. `gcc -Os -DREACH -DSAFETY pan.c -o datatrans` compiles the verifier (called pan (protocol analyser))

A bug in the protocol

./datatrans executes the generated verifier and gives:

```
pan: invalid end state (at depth 11)
pan: wrote datatrans.pml.trail

(Spin Version 5.2.2 -- 7 September 2009)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim           - (none specified)
assertion violations +
cycle checks         - (disabled by -DSAFETY)
invalid end states +
```


A deadlock in the protocol

Lets replay the error trail: `spin -c -t datatrans.pml`

```
proc 0 = Mproc
proc 1 = Wproc
q\p   0   1
  1   W!ini
  1   .   W?ini
  2   .   M!ack
  2   M?ack
  1   W!dreq
  1   .   W?dreq
  2   .   M!data
  2   M?data
  1   W!data
  1   .   W?data
spin: trail ends after 12 steps
```


A deadlock in the protocol (cont.)

```
-------------
final state:
-------------
#processes: 2
queue 2 (M):
queue 1 (W):
 12: proc  1 (Wproc) line  30 "datatrans.pml" (state 10)
 12: proc  0 (Mproc) line  16 "datatrans.pml" (state 11)
2 processes created
```


Suggested reading

- G. Holzmann (), Chapters 1–3
- (Baier & Katoen, 2008) Chapter 2
- http://spinroot.com/spin/Man/index.html
- Experiment with SPIN, try some models (see new Assignments)

Models of reactive processes

Reactive Systems

- We did not provide a definition yet but appealed to your intuition. Need to rectify this!
- We have a *formal* understanding of an algorithm (takes input and produces output upon termination)
- But most systems today are built to interact with a user (web servers, data bases, etc)
- We consider it to be an *error* if a reactive system terminates.

Definition

Reactive programs are programs whose purpose is to maintain an ongoing interaction with their environment, rather that produce a final value on termination.

The semantics of Promela

- Last week we introduced Promela and gave a very informal description of the semantics.
- Today, we open the box and look what is happening in SPIN.

Ordered set

Definition (Ordered Set)

Let S be a set and $\leq$ be a relation on S. $(S;\leq)$ is called a *ordered set*,

- if $\leq$ is transitive ($\forall s,t,u \in S : s < t \wedge t \leq u \implies s \leq u$)
- if $\leq$ is anti-symmetric ($\forall s,t \in S : s < t \wedge s \leq t \implies s = t$)

We call $(S;\leq)$ *totally ordered,* if and only if it is ordered and:

- $\forall s,t \in S : s \leq t \vee t \leq s$

Strictly Ordered set

Definition (Strictly Ordered Set)

Let S be a set and $<$ be a relation on S. $(S;<)$ is called a *strictly ordered set*, if $(S;<)$ is ordered and:

- $\forall a,b \in S : a < b \implies a \neq b$

Least element

Definition (Least element)

Let $(S;<)$ be an ordered set and let $T \subseteq S$ and $T \neq \emptyset$. We say that T has a least element, if there exists $s \in T$ such that for any $t \in T$ we have $s \leq t$.

Well-ordered set

Definition (Well-ordered set)

An ordered set $(S;\leq)$ is called *well-ordered,* if and only if any non-empty subset $T \subseteq S$ has a least element.

Intermission I: Ordinal numbers

- Ordinal numbers are naturally ordered numbers. They represent a very important mathematical structure when we talk about infinite structures.
- Algebraically and following the intuition, we have a total and well ordered set $(S, \leq)$ with least element 0 and a successor function $\cdot + 1 : S \to S$ such that $n < n+1$ for any $n \in S$.
- That only works for natural numbers and matches our understanding; let's look at the formal definition!

Intermission II: Ordinal numbers

Definition (von Neumann)

A set S is an ordinal if and only if S is *strictly* well-ordered with respect to set membership and every element of S is also a subset of S.

Example

Lets look at the first ordinals:

- $0 \triangleq \emptyset$
- $1 \triangleq \{\emptyset\} = \{0\}$ and note: $0 \in 1$ and $0 \subset 1$
- $2 \triangleq \{\emptyset, \{\emptyset\}\} = \{0, 1\}$ and note: $0 \in 2, 1 \in 2$
- $3 \triangleq \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}$ and so on ...

Intermission III: Ordinal numbers

- Observe that $\in$ and $\subset$ work as order relation: $2 = \{0, 1\}$ is in $3 = \{0, 1, 2\}$ and $2 \subseteq 3$.
- Define $n + 1 \triangleq n \cup \{n\}$ for every ordinal n. $n+1$ is an ordinal. Clearly, $n < n+1$ (because $n \in n+1$ and $n \subset n+1$.
- The natural numbers are isomorphic to the smallest set that includes $\emptyset$ and all successor ordinals of $\emptyset$.
- An important ordinal: ω is the *smallest* ordinal such that all natural numbers are in ω. It is also called the *smallest infinite ordinal*
- Theorem: Let $\omega + 1$ be the smallest ordinal that includes ω. Then $|\mathbb{N}| = |\omega| = |\omega + 1|$.
- The neat trick: We can now write $n \leq \omega$ to mean n is either a natural number or some countable infinite "thing".

Transition system

Definition

A *labelled transition system* T is a tuple $(S, A, \to, I, P, \mu)$, where

- S is a set of *states*,
- A is a set of *actions* that includes the special action τ,
- $\to \subseteq S \times A \times S$ a *transition relation*,
- $I \subseteq$ a set of *initial states*,
- P a set of *atomic propositions* and
- $\mu : S \to 2^P$ a *labelling function*.

The transition system T is called *finite* if S, A and P are finite.

Direct predecessor

Definition

Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system. Define for any $s \in S$ and $\alpha \in A$:

$$\mathrm{Pre}(s,\alpha) \triangleq \{s' \in S \mid s' \xrightarrow{\alpha} s\}$$
$$\mathrm{Pre}(s) \triangleq \bigcup_{\alpha \in A} \mathrm{Pre}(s,\alpha)$$
$$\mathrm{Post}(s,\alpha) \triangleq \{s' \in S \mid s \xrightarrow{\alpha} s'\}$$
$$\mathrm{Post}(s) \triangleq \bigcup_{\alpha \in A} \mathrm{Post}(s,\alpha)$$

Reading it out

- Each $s' \in \mathrm{Pre}(s,\alpha)$ is called a *direct α-predecessor of s*
- Each $s' \in \mathrm{Post}(s,\alpha)$ is called a *direct α-successor of s*
- Each $s' \in \mathrm{Pre}(s)$ is called a *direct predecessor of s*
- Each $s' \in \mathrm{Post}(s)$ is called a *direct successor of s*

Generalisation by *point-wise extension*

If $S' \subseteq S$, then define

$$\mathrm{Post}(S',\alpha) \triangleq \bigcup_{s \in S'} \mathrm{Post}(s,\alpha)$$

Likewise for other predecessors and successors.

Terminal state

Definition

Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system. A state $s \in S$ is called *terminal* if and only if $\mathrm{Post}(s) = \emptyset$.

Execution fragment

Definition
Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system. An *execution fragment* of length $n \leq \omega$ is an alternating sequence $s_0 \alpha_1 s_1 \alpha_2 s_2 \cdots \alpha_n s_n$ such that: $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $i < n$

Definition
An *initial execution fragment* is an execution fragment that starts in some initial state.

Definition
A *maximal execution fragment* is either finite and the last state is terminal or it is infinite.

Execution

Definition
Let T be a transition system. An *execution* is an initial and maximal execution fragment of T

Program graphs

Definition
A *program graph* P over a set V of (typed) variables is a tuple $(L, A, E, \hookrightarrow, I, g_0)$, where

- L is a set of locations and A is a set of actions.
- $E : A \rightarrow \Sigma \rightarrow \Sigma$ is the effect function.
- $\hookrightarrow \subseteq L \times 2^{\Sigma} \times A \times L$ is the conditional transition relation.
- $I \subseteq L$ is a set of initial locations
- $g_0 \in 2^{\Sigma}$ is the initial condition.

- Σ is a set of functions that assign values to variables.
- 2^{Σ} is the set of all predication on Σ.
- The notation $\ell \xhookrightarrow{[g]\alpha} \ell'$ represents a conditional transition.

Make sure our intuition remains valid

- The definition of a program graph is purely semantic!
- In principle, any statement can do anything and any predicate could represent anything. Don't do that!
- Guards and their notation: We write a formula φ (as we are used to) to represent the set $[\![\varphi]\!] \triangleq \{\sigma \in \Sigma \mid \sigma \models \varphi\}$ where $\sigma \models \varphi$ states that σ is a state in which φ is true.
- Define the *variant* $f[x \mapsto y]$ of a function f as:

$$f[x \mapsto y](z) \triangleq \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

- The effect of an assignment should represent the corresponding state changes, e.g.:

$$E(\mathtt{x}{=}\mathtt{1}) \triangleq \{(\sigma; \sigma[x \mapsto 1]) \mid \sigma \in \Sigma\}$$

Semantics of `skip`

$$\frac{}{\texttt{skip} \xhookrightarrow{[true]id} \text{exit}}$$

Semantics of assignment

$$\frac{}{\texttt{x=e} \xhookrightarrow{[true]assign(x,e)} \text{exit}}$$

Semantics of expression

$$\frac{}{\texttt{e} \xhookrightarrow{[e]id} \text{exit}}$$

Semantics of sequential composition

$$\frac{stmt_1 \xhookrightarrow{[g]e} \text{exit}}{stmt_1 ; stmt_2 \xhookrightarrow{[g]e} stmt_2}$$

Semantics of if statement

$$\frac{stmt_1 \xrightarrow{[g_i]e} stmt_2}{\textbf{if} \ldots :: stmt_1 :: \ldots \textbf{fi} \xrightarrow{[g_i]e} stmt_2}$$

Note that **else** represents the disjunction of the negation of every other guard: $\bigvee_{i<n} \neg g_i$.

Semantics of do statement

Every terminating statement can be embedded into a loop.

$$\frac{stmt_i \xrightarrow{[g_i]e} \text{exit}}{\textbf{do} \ldots :: stmt_i :: \ldots \textbf{od} \xrightarrow{[g_i]e} \textbf{do} \ldots :: stmt_i :: \ldots \textbf{od}}$$

A break statement exits the loop.

$$\frac{stmt_i \xrightarrow{[g_i]e} \textbf{break}}{\textbf{do} \ldots :: stmt_i :: \ldots \textbf{od} \xrightarrow{[g_i]e} \text{exit}}$$

In any other case execution may block...

$$\frac{stmt_i \xrightarrow{[g_i]e} stmt_j}{\textbf{do} \ldots :: stmt_i :: \ldots \textbf{od} \xrightarrow{[g_i]e} stmt_j; \textbf{do} \ldots :: stmt_i :: \ldots \textbf{od}}$$

Labels and go-to

Labels (`label:`) declare a node.
A go-to transfers control to the node.

$$\frac{}{\texttt{goto label} \xrightarrow{[true]id} \text{label}}$$

A note on `atomic`

The correct semantics of an atomic block is quite complicated, because we have to think about the internal expressions. If these are false, then the atomic block is interleaved. When these conditions are true, no interleaving is allowed. The tricky part is: we do not have a simple means of realising this in the transition system semantics.
SPIN uses coloured states to indicate possible interleaving points.
This means that atomic blocks are not always atomic.

A note on `d_step`

It is much simpler to provide a semantics for a **`d_step`**, because they must never be interleaved (and if they should, the system is considered to contain some error). This makes it possible to define a semantics!

For sake of simplicity, assume that the **`d_step`** does not contain any loops. First up, we must "massage" the contents of an atomic block into a certain format: the statement starts with one if-statement and each branch of the if-statement is followed by a sequence of assignments only (that can be done).

Then we can expand each branch, now in the form `g -> x = e; y = f; ...`, into one transition

```
if :: ... :: g -> x = e; y = f ... :: ... fi
```

$$\xhookrightarrow{[g]assign((x,y,\dots),(e,f,\dots))} \text{exit}$$

A note on test-and-set semantics

With reference to "Model checking" by (Baier & Katoen, 2008) and Katoen:

- (Baier & Katoen, 2008) and Katoen introduce a language they call "nanoPromela" which is supposed to be a simplification of Promela.
- NanoPromela does not include all features of Promela.
- It has a semantics based on program graphs and transition systems, just as explained above.
- For **`do..od`** and **`if..fi`** they propagate a "test-and-set" semantics, whereas we explain the semantics for Promela and Promela uses a "two-step" semantics.

Parallelism and communication

- For now, we just focus on interleaving semantics of transition systems and program graphs.
- The channel operators can be realised by compiling channels to shared variables (bounded buffers). This is actually what SPIN is doing; but it is also using tricks on optimising channels during model checking.

Interleaving of program graphs

Let $P_1 = (L_1, A_1, E_1, \hookrightarrow_1, l_1, g_{0,1})$ and $P_2 = (L_2, A_2, E_2, \hookrightarrow_2, l_2, g_{0,2})$ be program graphs over V_1 respectively v_2. Define

$$P_1 \parallel P_2 \triangleq (L_1 \times L_2, A_1 \uplus A_2, E, \hookrightarrow, l_1 \times l_2, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ is defined by

$$\frac{\ell_1 \xhookrightarrow{[g]a} \ell_1'}{(\ell_1,\ell_2) \xhookrightarrow{[g]a} (\ell_1',\ell_2)} \quad \text{and} \quad \frac{\ell_2 \xhookrightarrow{[g]a} \ell_2'}{(\ell_1,\ell_2) \xhookrightarrow{[g]a} (\ell_1,\ell_2')}$$

where $S \uplus S'$ represents disjoint union of the set S and S' and E is defined by:

$$E(a) \triangleq \begin{cases} E_1(a) & \text{if } a \in A_1 \\ E_2(a) & \text{if } a \in A_2 \end{cases}$$

Interleaving of transition systems

Let $T_1 = (S_1, A_1, E_2, \rightarrow_1, I_1, P_1, \mu_1)$ and $T_2 = (S_2, A_2, E_2, \rightarrow_2, I_2, P_2, \mu_2)$ be program graphs over V_1 respectively v_2. Define

$$T_1 \quad \| \quad T_2 \quad = \quad (S_1 \quad \times \quad S_2, A_1 \quad \cup \quad A_2, \rightarrow, I_1 \quad \times \quad I_2, P_1 \quad \cup \quad P_2, \mu)$$

where $\hookrightarrow$ is defined by

$$\frac{s_1 \xrightarrow{a} s_1'}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \quad \text{and} \quad \frac{s_2 \xrightarrow{a} s_2'}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')}$$

and

$$\mu((s_1, s_2)) = \mu_1(s_1) \cup \mu_2(s_2)$$

Observers

Observers I

- One generic way of specifying properties of systems is by *observers*
- An observer is a *process* that raises an error once it observes that a property has been violated.
- Observers run in lock-step with all other processes (technically, not interleaving)
- In SPIN, they are formalised by `never` claims
- More generically, they can be described by processes
- Observers are also important during *run-time verification* (later)

Observers II

- We have seen some principles in model checking, already:
  - Building a transition system from a program graph
  - Using *exhaustive state exploration* for assertion checking
  - Checking for deadlocks (criterion: there is a state in the transition system without outgoing transition, *but* the program graph contains one outgoing transition)
- How do we check “absence of starvation”? Lets go step by step...

`never` claim

Instead of an assertion, once can use an observer like this:

```
never {
do
:: (in_cs > 1) -> break
:: else
od
}
```

It is an error, if a never claim automaton terminates, thus this automaton tests for mutual exclusion.

`never` claims are Büchi automatons

- A Büchi automaton is a finite state automaton that accepts infinite words
- We call the language accepted by a Büchi automaton ω-regular
- There are also ω-regular expressions

Recall regular expressions and finite automata

- A regular expression r has the form $a \mid r+r \mid r \cdot r \mid r^* \mid \bar{r}$
- A string that matches a regular expression can be recognised by a finite automaton

ω-regular expressions

- A ω-regular expression r has the form $r ::= r+r \mid s \cdot s^\omega$, where $s ::= a \mid s+s \mid s \cdot s \mid s^* \mid \bar{s}$
- A string that matches a ω-regular expression can be recognised by a finite automaton with *Büchi acceptance condition*, or *Büchi automaton*

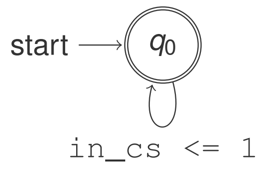
Definition

Definition

A *non-deterministic Büchi automaton* (NBA) A is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set of states,
- Σ is an alphabet
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function
- $Q_0 \subseteq Q$ is a set of initial states
- $F \subseteq Q$ is a set of accepting (or final) states, called *acceptance set*

The invariance automaton



Run

Definition

A *run* for $\sigma \in \Sigma^\omega$ is an infinite sequence $\alpha \in Q^\omega$ such that $\alpha_0 \in Q_0$ and for all $i < \omega$: $\alpha_{i+1} \in \delta(\alpha_i, \sigma_i)$.

Definition

A run α is accepting, if there is an infinite set of indexes $I \subseteq \omega$ such that all $\alpha_i \in F$ where $i \in I$.

Remark

Alternatively: there is an injective function $f : \omega \to \omega$ such that $\alpha_{f(i)} \in F$

Stutter extension

- With this notion of ω-properties, we cannot establish properties of *finite* runs.
- By *repeating* the final state infinitely often, we can transform any finite run to an infinite one.
- That is, if ε represents a *stutter step*, any run σ of length n can be extended to an infinite run by:

$$\sigma \cdot (\sigma_n, \varepsilon, \sigma_n)^\omega$$

Finite states, infinite runs

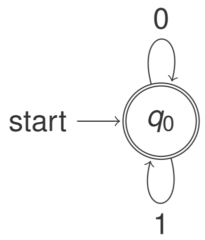


Figure: How many different strings does this accept?

We can specify *fairness*

```
never {
Q0_init:
        if
        :: (flag[0]) -> goto accept_Q1
        :: (1) -> goto Q0_init
        fi;
accept_Q1:
        (flag[0]) -> goto accept_Q1
}
```

- By marking a location with a label that starts with `accept`, we place its state into the acceptance set.

The automaton

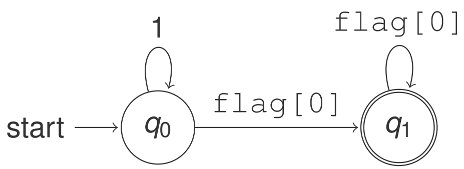


Figure: An Büchi automaton to *refute* fairness

How does the observer work?

- The observer represents a Büchi automaton whose alphabet are propositions.
- After each step taken by some process, the observer automaton takes a step
- If the observer terminates, SPIN flags an error
- If the run accepts, SPIN flags an error.

In SPIN, observers reduce the search space

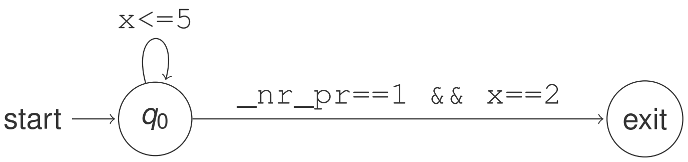


Figure: What does this one check?

```
never {
  do :: (x <= 5)
     :: (_nr_pr == 1 && x == 2) -> break
  od
}
```


Composition rule

Definition

Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system and $\mathscr{A} = (Q, \Sigma, \delta, Q_0, F)$ a Büchi automaton. Define the product $T \otimes \mathscr{A} \triangleq (S \times Q, A, \rightharpoondown, I', Q, \mu')$ where $\rightharpoondown$ is defined by:

$$\frac{s \xrightarrow{a} t \qquad p \xrightarrow{\mu(t)} q}{(s,p) \overset{a}{\rightharpoondown} (t,q)}$$

and where

- $I' = \{(s_0, q) \mid s_0 \in I \wedge \exists q_0 \in Q : q = \delta(q_0, \mu(s_0))\}$
- $\mu' : S \times Q \rightarrow 2^Q$ is given by $\mu'((s,q)) = \{q\}$.

Let $P_{pers(\mathscr{A})}$ be the persistence property "eventually forever $\neg F$, where $\neg F \triangleq \bigwedge_{q \in F} \neg q$.

Verification of ω-regular properties

Theorem

Let T be a transition system without terminal states over P and let $\mathscr{P}$ be an ω-regular property over P. Furthermore, let $\mathscr{A}$ be a non-blocking NBA with the alphabet 2^P and $\mathscr{L}(\mathscr{A}) = (2^P)^\omega \setminus P$. Then the following statements are equivalent:

1. $T \models P$
2. $[\![T]\!] \cap \mathscr{L}(\mathscr{A}) = \emptyset$
3. $T \otimes \mathscr{A} \models P_{pers(\mathscr{A})}$

Observation

- The composition rule forms some kind of *intersection* between the runs described by the transition system and the language accepted by the Büchi automaton.
- In SPIN, this can be used to *reduce* the states we search in.
- We must think carefully, because we can *remove* a path to an error state with the same mechanism.

Classification of properties

- Invariants
- Safety properties
- Liveness properties
- Other properties

Invariants

Definition

An *invariant* φ is a property such that:

$$P_{inv} = \{\sigma \in 2^{P^{\omega}} \mid \forall i < \omega : \sigma_i \models \varphi\}$$

In short: an invariant holds in every state.

Safety property

"Never something bad should happen"

Definition

A property P_{safe} is called a *safety property* if for all words σ there exists a finite prefix $\hat{\sigma}$ such that

$$P_{safe} \cap \{\sigma' \in (2^P)^{\omega} \mid \exists \hat{\sigma} : \hat{\sigma} \text{ is a finite prefix of } \sigma'\} = \emptyset$$

$\hat{\sigma}$ is called *bad prefix*

A bad prefix is a finite refutation of the safety property. The defining intuition is: any safety property can be refuted in finite time (the bad thing happened)

Liveness property

"Eventually something good happens"

Definition

A liveness property is a property that admits any prefix, i.e., it has the form $(2^P)^* \cdot r^{\omega}$ for some regular expression r.

Observe: A liveness property cannot be refuted by any finite prefix.

Fairness

Unconditional fairness "Every process gets its turn infinitely often"

Weak fairness "Every process that is continuously enabled from a certain time instant on gets its turn infinitely often"

Strong fairness "Every process that is infinitely often enabled gets its turn infinitely often"

SPIN and `progress`

- Sometimes we need to *assume* fairness to verify a property.
- We can specify fairness as part of the never claim, thus ruling out all unfair computations
- We can also use `progress` labels.
  - We use progress labels to mark statements that do something desirable.
  - When a process contains a progress label, we can instruct the verifier to make sure that all runs visit some progress label infinitely often.
  - This way, we can force *fairness*
- Finally, we will return to fairness later.

Linear-time temporal logic

What I might remember, what I need to know ...

- In every formal system, we have to distinguish a couple of closely related concepts.
- Implication is a concept encoded in a logical formula, usually written $a \implies b$.
- Provability is a concept of the proof system of a logic. It is usually written $a \vdash b$ and means "assuming a I can prove b".
- Satisfaction connects models with formulas: $M \models a$ says that M is a semantic model in which a is true.
- Entailment is an implication on the semantic level: $a \Vdash b$ states: in every model in which a holds, also b will hold.
- The three relations $\implies$, $\vdash$, $\Vdash$ are closely related in first order logic, but there are logics which behave differently.

Deduction lemma

Lemma

Let Φ be a set of formulas and φ, ψ formulas such that $\Phi \cup \{\varphi\} \vdash \psi$. Then $\Phi \vdash \varphi \implies \psi$.

This lemma connects provability with implication by: When I can prove a sentence ψ by assuming Φ and φ, then I can also prove $\varphi \implies \psi$ with only assuming Φ.

The reverse is usually encoded by the rule *modus ponens*:

$$\frac{\varphi \qquad \varphi \implies \psi}{\psi}$$

Soundness and completeness

- Soundness means that one can only prove true sentences with the proof system, i.e.: $\Phi \vdash \Psi$ implies $\Phi \Vdash \Psi$.
- Completeness means that every true sentence is provable, i.e.: $\Phi \Vdash \Psi$ implies $\Phi \vdash \Psi$.
- Existence of a proof does not mean that we can find it or that a computer can find it!
- Decidability: $\Phi \overset{?}{\vdash} \Psi$ can be *constructively* answered by finding the proof. One can, in principle, enumerate all proofs and check whether it proves the claim. That is not always possible.

Temporal logics

- Often, observers are not really convenient, when we reason about models and specify their properties.
- A lot of properties can be structures quite well when we express them in a special logic.
- One such family of logics is called *temporal logic*
- Some temporal logic formulas can be translated into never claims (even by SPIN)

Assertions

Definition

An *assertion* is a predicate that may hold in some state.

Example

`in_cs < 2` is an assertion, because we only need knowledge about the current state to determine the validity of this formula.

Talking about time

Definition

A *path* formula is constructed by the following rules:

- Every assertion is a path formula
- If φ is a path formula, then $\bigcirc\varphi$ is a path formula (read "next φ")
- If φ is a path formula, then $\Box\varphi$ is a path formula (read "always φ")
- If φ is a path formula, then $\Diamond\varphi$ is a path formula (read "eventually φ")
- If φ and ϕ are path formulas, then $\varphi\mathscr{U}\psi$ is a path formula (read "φ holds until ψ")
- If φ and ϕ are path formulas, then $\varphi\mathscr{R}\psi$ is a path formula (read "φ releases ψ")

Talking about time

Definition

A *path* formula is constructed by the following rules:

- If φ is a path formula, then $\neg\varphi$ is a path formula (read "not φ")
- If φ and ϕ are path formulas, then $\varphi \wedge \psi$ is a path formula (read "φ and ψ")
- There are no other path formulas (but we may introduce more abbreviations, e.g. $\vee$)

Paths

To interpret these formulas, we need a *path,* which we can derive from an *execution (fragment)*.
Recall: An execution fragment of length $n \leq \omega$ is an alternating sequence $s_0\alpha_1 s_1\alpha_2 s_2\cdots\alpha_n s_n$. The *path* is the sequence $s_0 s_1 s_2\cdots s_n$.
Let φ be an assertion. A path $\sigma = s_0 s_1 s_2\cdots s_n$ satisfies φ, written $\sigma \models \varphi$, if and only if φ holds in the first state of σ, that is: $s_0 \models \varphi$.

The meaning of next

Definition

The *first derivative* of a sequence $\alpha = s_0 s_1 s_2\cdots s_n$ is α without its first element, i.e. $s_1 s_2\cdots s_n$. We write α' or α^1 for the first derivative.
We can define the *n*th derivative inductively: If $n = 0$, then $\alpha^n = \alpha$ and if $n = 1$ then $\alpha^n = \alpha'$. If $n > 1$, then $\alpha^n = \alpha^{n-1'}$.

Definition

$\bigcirc\varphi$ reads φ holds in the next state. Let σ be a path. $\sigma \models \bigcirc\varphi$ iff $\sigma^1 \models \varphi$.

Example

Let $\sigma = (x = 0)(x = 1)(x = 2)(x = 3)\ldots$.

- $\sigma \models x = 0$, $\sigma \not\models x = 1$
- $\sigma \not\models \bigcirc x = 0$, $\sigma \models \bigcirc x = 1$
- $\sigma \models \bigcirc\bigcirc x = 2$

The meaning of always

Two equivalent formulations:

Definition
$\sigma \models \Box\varphi$ if and only if $\forall i : \sigma^i \models \varphi$

Definition
$\sigma \models \Box\varphi$ if and only if $\sigma \models \varphi$ and $\sigma' \models \Box\varphi$

Lemma
$\forall i : \sigma^i \models \varphi$ if and only if $\sigma \models \varphi$ and $\sigma' \models \Box\varphi$

The meaning of eventually

Two equivalent formulations:

Definition
$\sigma \models \Diamond\varphi$ if and only if $\exists i : \sigma^i \models \Diamond\varphi$

Definition
$\sigma \models \Diamond\varphi$ if and only if $\sigma \models \varphi$ or $\sigma' \models \Diamond\varphi$

Lemma
$\exists i : \sigma^i \models \varphi$ if and only if $\sigma \models \varphi$ or $\sigma' \models \varphi$

Lemma
$\Diamond\varphi \iff \neg\Box\neg\varphi$

The meaning of until

Definition
$\sigma \models \varphi \mathscr{U} \psi$ if and only if $\exists i : \sigma^i \models \psi \wedge \forall j < i : \sigma^j \models \varphi$

Lemma

- $\Diamond\varphi \iff \mathbf{true}\mathscr{U}\varphi$

The meaning of release

Definition
$\sigma \models \varphi \mathscr{R} \psi$ if and only if $(\forall i : \sigma^i \models \psi) \vee (\exists i : \sigma^i \models \varphi \wedge \forall j < i : \sigma^j \models \psi)$

Lemma

- $\Box\varphi \iff \varphi\mathscr{R}\mathbf{false}$
- $\neg(\varphi\mathscr{U}\psi) \iff \neg\varphi\mathscr{R}\neg\psi$

Useful equivalences I

Duality laws

$$\neg \bigcirc \varphi \equiv \bigcirc \neg \varphi$$
$$\neg \Diamond \varphi \equiv \Box \neg \varphi$$
$$\neg \Box \varphi \equiv \Diamond \neg \varphi$$
$$\neg(\varphi \mathcal{U} \psi) \equiv ((\varphi \wedge \neg \psi) \mathcal{U} (\neg \varphi \wedge \neg \psi)) \vee \Box(\varphi \wedge \neg \psi)$$

Idempotency laws

$$\Diamond\Diamond\varphi \equiv \Diamond\varphi$$
$$\Box\Box\varphi \equiv \Box\varphi$$
$$\varphi \mathcal{U} (\varphi \mathcal{U} \psi) \equiv \varphi \mathcal{U} \psi$$
$$(\varphi \mathcal{U} \psi) \mathcal{U} \psi \equiv \varphi \mathcal{U} \psi$$

Useful equivalences II

Absorption laws

$$\Diamond\Box\Diamond\varphi \equiv \Box\Diamond\varphi$$
$$\Box\Diamond\Box\varphi \equiv \Diamond\Box\varphi$$

expansion laws

$$\varphi \mathcal{U} \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \mathcal{U} \psi))$$
$$\Diamond\varphi \equiv \varphi \vee (\bigcirc\Diamond\varphi)$$
$$\Box\varphi \equiv \varphi \wedge (\bigcirc\Box\varphi)$$

distributive laws

$$\bigcirc(\varphi \mathcal{U} \psi) \equiv (\bigcirc\varphi \mathcal{U} \bigcirc \psi)$$
$$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$$
$$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$$

Suggested reading

– (Baier & Katoen, 2008), 3.3–3.5, 5.1
– Holzmann, Chapter 4

The basic steps

1. Translate an LTL formula into a Büchi automaton
2. Compute the product of the Büchi automaton and the system
3. Check, whether the language accepted by the product is empty

Chart

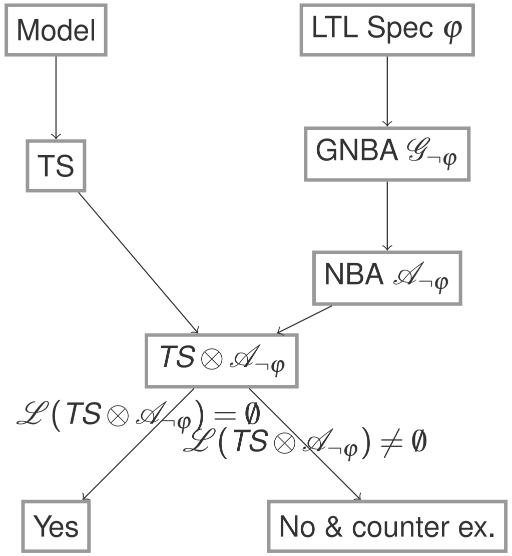


Figure: Flow chart

Telematics Computer Systems
Automatons-based model checking
LTL model checking algorithm
Freie Universität Berlin

A bit of mathematical motivation

- We want to show: $M \models \varphi$
- This means $[\![M]\!] \subseteq [\![\varphi]\!]$
- This holds, when $[\![M]\!] \cap (\Sigma^\omega \setminus [\![\varphi]\!]) = \emptyset$
- $[\![M]\!]$ is already represented as a transition system
- For any φ we can construct a Büchi automaton A_φ that accepts any trace that is a model of φ
- We know how to build $M \oplus A_\varphi$, but can we answer $[\![M \oplus A_\varphi]\!] \stackrel{?}{=} \emptyset$

Telematics Computer Systems
Automatons-based model checking
LTL model checking algorithm
Freie Universität Berlin

Deciding emptiness

Lemma

Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. Then the following two statements are equivalent:

1. $[\![A]\!] \neq \emptyset$
2. *There exists a $q \in F$ that is reachable from some $q_0 \in Q_0$ such that there is a cycle from q to q in A.*

Telematics Computer Systems
Automatons-based model checking
LTL model checking algorithm
Freie Universität Berlin

The algorithm

- Compute the strongly connected components S of $(Q, \hat{\delta})$ that is the directed graph obtained from the NBA.
- Generate $\hat{\mathcal{S}} \triangleq \{S \in \mathcal{S} \mid S \cap F \neq \emptyset\}$
- For each $q_0 \in Q_0$ search for a path from q_0 to some member of $\hat{S} \in \hat{\mathcal{S}}$.
- If some path has been found, print out the path and the members of $\hat{S}$ to indicate the acceptance cycle.

From LTL to NBA

Next, we fill in the second part of the equation.
Useful properties:

Lemma
$\varphi \mathscr{R} \psi \equiv \neg(\neg\varphi \mathscr{U} \neg\psi)$

Lemma
$\Box\varphi \equiv \neg(true \mathscr{U} \neg\varphi)$

Lemma
$\varphi \mathscr{U} \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \mathscr{U} \psi))$

Theorem
A complete and minimal set of operators to express all possible LTL formulae is $\neg, \wedge, \mathscr{U}, \bigcirc$, true.

Telematics Computer Systems — Automatons-based model checking — LTL model checking algorithm — Freie Universität Berlin

Generalised Büchi automaton

Definition
A generalised non-deterministic Büchi automaton (GNBA) is a tuple $\mathscr{G} \triangleq (Q, \Sigma, \delta, Q_0, \mathscr{F})$, where Q, Σ, δ, Q_0 are defined as for NBA and $\mathscr{F} \subseteq 2^Q$ is a possibly empty set of *acceptance sets*.
The accepted language $\mathscr{L}(\mathscr{G})$ consists of all $\sigma \in (2^P)^\omega$ that have a run in $\mathscr{G}$ such that:

$$\forall F \in \mathscr{F} : \exists f \in \omega \to \omega : \forall i < \omega : \sigma(f(i)) \in F$$

Telematics Computer Systems — Automatons-based model checking — LTL model checking algorithm — Freie Universität Berlin

From GNBA to NBA

Theorem
For each GNBA G there exists an NBA A with $\mathscr{L}(G) = \mathscr{A}$ and $|A| = O(|G| \cdot |\mathscr{F}|)$ where $\mathscr{F}$ denotes the acceptance set of G.

Proof.
Let $G = (Q, \Sigma, \delta, Q_0, \mathscr{F})$ be a GNBA. Assume w.l.o.g. $\mathscr{F} \neq \emptyset$. Let $\mathscr{F} = \{F_0, \dots, F_{k-1}\}$. The basic idea is to create k copies of A where the acceptance set F_i of the ith copy is connected to the successor states in the $i+1$th copy. The acceptance set of the NBA becomes F_0. □

Telematics Computer Systems — Automatons-based model checking — LTL model checking algorithm — Freie Universität Berlin

Proof continued

Formally, define $Q' = Q \times \{0, \dots, k-1\}$, $Q'_0 = Q_0 \times \{0\}$, $F' = F_1 \times \{0\}$. The transition function is given by

$$\delta'(\langle q, i\rangle, A) = \begin{cases} \{\langle q', i\rangle \mid q' \in \delta(q, A)\} & \text{if } q \notin F_i \\ \{\langle q', (i+1) \bmod k\rangle \mid q' \in \delta(q, A)\} & \text{otherwise} \end{cases}$$

Proving $\mathscr{L}(A) \subseteq \mathscr{L}(A)$: For a run in A to be accepting, it must visit states in F_0 infinitely often. Since all successors of states in F_i are in the $i+1$th copy, we have to visit members of F_{i+1} k times before we can visit members of F_i for the $k+1$th time. Hence, states for all F_i are visited infinitely often.
Proving $\mathscr{L}(G) \subseteq \mathscr{L}(G)$ uses similar reasoning.

Constructing a GNBA for a LTL formula φ

Assume that φ only contains the operators $\wedge$, $\neg$, $\bigcirc$, and $\mathcal{U}$. Since $\varphi = \mathit{true}$ is trivial, we also assume $\varphi \neq \mathit{true}$.
Basic idea:
Let $\sigma = A_0 A_1 A_2 \ldots \models \varphi$. The sets A_i are expanded by subformulae (and their negations) ψ of φ such that an infinite word $\hat{\sigma} = B_0 B_1 B_2 \ldots$ with the following property arises:

$$\psi \in B_i \text{ if and only if } \sigma^i \models \psi$$

Closure

The set of all subformulae and their negations (where we identify $\neg\neg\psi$ with ψ) of a formula is called its *closure*. It is defined by:

$$\begin{aligned}
\mathit{closure}(a) &\triangleq \{a, \neg a\} \\
\mathit{closure}(\neg\varphi) &\triangleq \{\neg\varphi, \varphi\} \cup \mathit{closure}(\varphi) \\
\mathit{closure}(\bigcirc\varphi) &\triangleq \{\bigcirc\varphi, \neg\bigcirc\varphi\} \cup \mathit{closure}(\varphi) \\
\mathit{closure}(\varphi \wedge \psi) &\triangleq \{\varphi \wedge \psi, \neg(\varphi \wedge \psi)\} \cup \mathit{closure}(\varphi) \cup \mathit{closure}(\psi) \\
\mathit{closure}(\varphi \,\mathcal{U}\, \psi) &\triangleq \{\varphi \,\mathcal{U}\, \psi, \neg(\varphi \,\mathcal{U}\, \psi)\} \cup \mathit{closure}(\varphi) \cup \mathit{closure}(\psi)
\end{aligned}$$

Elementary set of formulae

We cannot use all subsets of a closure, but only elementary ones. Let $B \subseteq \mathit{closure}(\varphi)$.

1. B is *consistent* wrt. propositional logic, i.e. for all $\varphi_1 \wedge \varphi_2, \psi$:
   - $\varphi_1 \wedge \varphi_2 \in B \iff \varphi_1 \in B$ and $\varphi_2 \in B$
   - $\psi \in B \implies \neg\psi \notin B$
   - $\mathit{true} \in \mathit{closure}(\varphi) \implies \mathit{true} \in B$
2. B is *locally consistent* wrt. until, i.e. for all $\varphi_1 \,\mathcal{U}\, \varphi_2 \in \mathit{closure}(\varphi)$:
   - $\varphi_2 \in B \implies \varphi_1 \,\mathcal{U}\, \varphi_2 \in B$
   - $\varphi_1 \,\mathcal{U}\, \varphi_2 \in B$ and $\varphi_2 \notin B \implies \varphi_1 \in B$
3. B is maximal, i.e., for all $\psi \in \mathit{closure}(\varphi)$:
   - $\psi \notin B \implies \neg\psi \in B$

GNBA for LTL formula

Theorem

For every LTL formula φ there exists a GNBA G_φ such that:

- $[\![\varphi]\!] = \mathcal{L}(G_\varphi)$
- *G_φ can be constructed in time and space $2^{O(|\varphi|)}$*
- *The number of accepting sets of G_φ is bounded above by $O(|\varphi|)$*

Construction

Define $G_\varphi = (Q, 2^P, \delta, Q_0, \mathscr{F})$ by:

- Q is the set of all elementary sets of formulae $B \subseteq closure(\varphi)$
- $Q_0 = \{B \in Q \mid \varphi \in B\}$
- $\mathscr{F} = \{F_{\varphi_1 \mathscr{U} \varphi_2} \mid \varphi_1 \mathscr{U} \varphi_2 \in closure(\varphi)\}$ where $F_{\varphi_1 \mathscr{U} \varphi_2} = \{B \in Q \mid \varphi_1 \mathscr{U} \varphi_2 \notin B \text{ or } \varphi_2 \in B\}$
- The transition relation δ is given by:
  - If $A \neq B \cap P$, then $\delta(B, A) = \emptyset$
  - If $A = B \cap P$, then $\delta(B, A)$ is the set of all elementary sets of formulae B' satisfying:
    1. For every $\bigcirc \psi closure(\varphi)$: $\bigcirc \psi \in B \iff \psi \in B'$
    2. For every $\varphi_1 \mathscr{U} \varphi_2 closure(\varphi)$: $\varphi_1 \mathscr{U} \varphi_2 \in B \iff \varphi_2 \in B$ or $(\varphi_1 \in B$ and $\varphi_1 \mathscr{U} \varphi_2 \in B')$

Proof I

$[\![\varphi]\!] \subseteq \mathscr{L}(G)$: Let $\sigma \in [\![\varphi]\!]$. Then $\sigma \in (2^P)^\omega$ and $\sigma \models \varphi$. Define the elementary set $B^i = \{\psi \in closure(\varphi) \mid \sigma^i \models \psi\}$. Then, obviously $B_i \in Q$. Now we prove that $B_0 B_1 \ldots$ is an accepting run of G for σ. Observe $B_{i+1} \in \delta(B_i, \sigma(i))$, because for all i:

Proof II: $B_0 B_1 \ldots$ is a run of G

- $\sigma(i) = B_i \cap P$
- for $\bigcirc \psi \in closure(\varphi)$:

$$\begin{aligned} & \bigcirc \psi \in B_i \\ \text{iff } & \sigma^i \models \bigcirc \psi \\ \text{iff } & \sigma^{i+1} \models \psi \\ \text{iff } & \psi \in B_{i+1} \end{aligned}$$

- for $\varphi_1 \mathscr{U} \varphi_2 \in closure(\varphi)$:

$$\begin{aligned} & \varphi_1 \mathscr{U} \varphi_2 \in B_i \\ \text{iff } & \sigma^i \models \varphi_1 \mathscr{U} \varphi_2 \\ \text{iff } & \sigma^i \models \varphi_2 \text{ or } (\sigma^i \models \varphi_1 \text{ and } \sigma^{i+1} \models \varphi_1 \mathscr{U} \varphi_2) \\ \text{iff } & \varphi_2 \in B_i \text{ or } (\varphi_1 \in B_i \text{ and } \varphi_1 \mathscr{U} \varphi_2 \in B_{i+1}) \end{aligned}$$

Proof III: The run is accepting

To prove: for each subformula $\varphi_1 \mathscr{U} \varphi_2 \in closure(\varphi)$, $B_i \in F_{\varphi_1 \mathscr{U} \varphi_2}$ infinitely often. Proof by contradiction: Assume there are finitely many i such that $B_i \in F_{\varphi_1 \mathscr{U} \varphi_2}$. We have:
$B_k \notin F_{\varphi_1 \mathscr{U} \varphi_2} \implies \varphi_1 \mathscr{U} \varphi_2 \in B_k$ and $\varphi_2 \notin B_k$
As $B_k = \{\psi \in closure(\varphi) \mid \sigma^k \models \psi\}$, it follows that if $B_k \notin F_j$, then:
$\sigma^k \models \varphi_1 \mathscr{U} \varphi_2$ and $\sigma^k \not\models \varphi_2$ Thus, $\sigma^l \models \varphi_2$ for some $l > k$. By definition of B_k, it then follows that $\varphi_2 \in B_l$ and by definition of F_j, $B_k \in F_j$. Thus, $B_i \in F_j$ for finitely many i, then $B_k \in F_j$ for infinitely many k. Contradiction.

Proof IV

$\mathscr{L}(G) \subset [\![\varphi]\!]$: Let $\sigma = A_0 A_1 \ldots \in \mathscr{L}(G)$. Then there is an accepting run $B_0 B_1 \ldots$ for σ in G.
Since $\delta(B, A) = \emptyset$ for all B and A with $A \neq B \cap P$, it follows that $A_i = B_i \cap P$ for all i. Thus $\sigma = (B_0 \cap P)(B_1 \cap P)(B_2 \cap P) \ldots$. To prove now:
$(B_0 \cap P)(B_1 \cap P)(B_2 \cap P) \ldots \models \varphi$. We prove the more general proposition:
For $B_0 B_1 B_2 \ldots$ a sequence with $B_i \in Q$ satisfying

1. for all i: $B_{i+1} \in \delta(B_i, A_i)$ and
2. for all $F \in \mathscr{F} : \exists^\infty j : B_j \in F$

we have for all $\psi \in closure(\varphi)$:

$$\psi \in B_0 \iff A_0 A_1 A_2 \models \psi$$

Proof V

By structural induction:
Base case: The statement for $\psi = true$ of $\psi = a$ with $a \in P$ follows directly from the definition of B_0 and the definition of closure.
Induction step: Based on the induction hypothesis that the claim holds for $\psi', \varphi_1, \varphi_2 \in closure(\varphi)$ we prove that for the formulae $\psi = \bigcirc \psi'$, $\psi = \neg \psi'$, $\psi = \varphi_1 \wedge \varphi_2$ and $\psi = \varphi_1 \mathscr{U} \varphi_2$ the claim also holds.
We only consider the (difficult) case $\varphi_1 \mathscr{U} \varphi_2$: Let $A_0 A_1 A_2 \in (2^P)^\omega$ and $B_0 B_1 B_2 \ldots \in Q^\omega$ satisfying the constraints 1 and 2. It is now shows that: $\psi \in B_0$ if and only if $A_0 A_1 A_2 \ldots \models \psi$.

Proof $\Rightarrow$

Assume $A_0 A_1 A_2 \ldots \models \varphi_1 \mathscr{U} \varphi_2$. Then there exists j such that $A_j A_{j+1} \ldots \models \varphi_2$ and $A_i A_{i+1} \ldots \models \varphi_1$ for $0 \leq i < j$. From the induction hypothesis it follows that $\varphi_2 \in B_j$ and $\varphi_1 \in B_i$ for $0 \leq i < j$. By induction on j we obtain $\varphi_1 \mathscr{U} \varphi_2 \in B_j, B_{j-1} \ldots B_0$.

Proof $\Leftarrow$

Assume $\varphi_1 \mathscr{U} \varphi_2 \in B_0$. Since B_0 is elementary, $\varphi_1 \in B_0$ or $\varphi_2 \in B_0$.
Case $\varphi_2 \in B_0$: From the induction hypothesis it follows $A_0 A_1 A_2 \ldots \models \varphi_2$ and thus $A_0 A_1 A_2 \ldots \models \varphi_1 \mathscr{U} \varphi_2$.
Case $\varphi_2 \notin B_0$. Then $\varphi_1 \in B_0$ and $\varphi_1 \mathscr{U} \varphi_2 \in B_0$. Assume $\varphi_2 \notin B_j$ for all j.
From the definition of δ we obtain using an inductive argument (successively applied to $\varphi_1 \in B_j, \varphi_2 \notin B_j$ and $\varphi_1 \mathscr{U} \varphi_2 \in B_j$ for all j):

$$\varphi_1 \in B_j \text{ and } \varphi_1 \mathscr{U} \varphi_2 \in B_j \text{ for all } j$$

Because $B_0 B_1 \ldots$ satisfies constraint 2, it follows that

$$B_j \in F_{\varphi_1 \mathscr{U} \varphi_2} \text{ for infinitely many } j$$

On the other hand:

$$(\varphi_2 \notin B_j \text{ and } \varphi_1 \mathscr{U} \varphi_2 \in B_j) \text{ iff } B_j \notin F_{\varphi_1 \mathscr{U} \varphi_2}$$

for all j. Contradiction!

Thus, we find a smallest j with $\varphi_2 \in B_j$, i.e. $\varphi_2 \notin B_0 B_1 \dots B_{j-1}$. The induction hypothesis for $i < j$ yields:

$$\varphi_1 \in B_i \text{ and } \varphi_1 \mathcal{U} \varphi_2 \in B_1 \text{for all } i < j.$$

Thus, $A_j, A_{j+1} \dots \models \varphi_2$ and $A_i A_{i+1} \dots \not\models \varphi_2$ for all $i < j$. We conclude that $A_0 A_1 A_2 \dots \models \varphi_1 \mathcal{U} \varphi_2$.
Q.E.D.

How difficult is model checking?

- How difficult is model checking?
- The answer is: very difficult.
- We finish the discussion of LTL model checking by mentioning the most important results about its complexity.

Reminder on complexity

P A problem is in *P*, if and only if there exists a deterministic algorithm that solves it in polynomial time.

NP A problem is in *NP*, if and only if there exists a non-deterministic algorithm that solves it in polynomial time.

PSPACE A problem is in *PSPACE*, if and only if there exists a deterministic algorithm that solves it in polynomial space.

Lemma
$P \subseteq NP \subseteq PSPACE$

Remark
The suspicion is: $P \subset NP \subset PSPACE$

Completeness: A problem P is complete for a complexity class C if any problem $P' \in C$ can be reduced to P.

Checking language emptiness

Theorem
There is an algorithm ("nested depth-first search") that needs $O(N + M + N \cdot |\Phi|)$ steps, where N is the number of reachable states, M is the number of transitions between reachable states, and Φ is some complex formula we have to check for the state.

Example
The "complex formula" might be the persistence property mentioned before.

Complexity results

Theorem

The LTL model-checking problem is PSPACE-complete.

Proof: See Theorem 5.48 in (Baier & Katoen, 2008).

Theorem

The satisfiability and validity problems for LTL are PSPACE-complete.

Proof: See Theorem 5.49 in (Baier & Katoen, 2008).

Fairness

Motivating examples

Consider the following simple mutual exclusion algorithm using semaphores:

```
inline P(s) { atomic { (s > 0) -> s-- } }
inline V(s) { atomic { s++ } }
byte in_cs, sema = 1;
active [2] proctype proc() {
  Remainder:
    P(sema);
  Critical:
    V(sema);
    goto Remainder;
}
```


Which of the desired properties hold?

1. Safety (mutual exclusion)?
2. Deadlock freedom?
3. Lifeness (absence of starvation)?

Lets prove some.

SPIN predicates and labels

- Internally, spin maintains an array of locations in which each process resides.
- `P[0]@Critical` means that the process with `_pid` 0 is currently in a line labelled by `Critical`.
- The following specification is dangerous, since we assume that the pids are 0 and 1.

$$\Box\neg(P[0]@\mathit{Critical} \wedge P[0]@\mathit{Critical})$$

is true, but

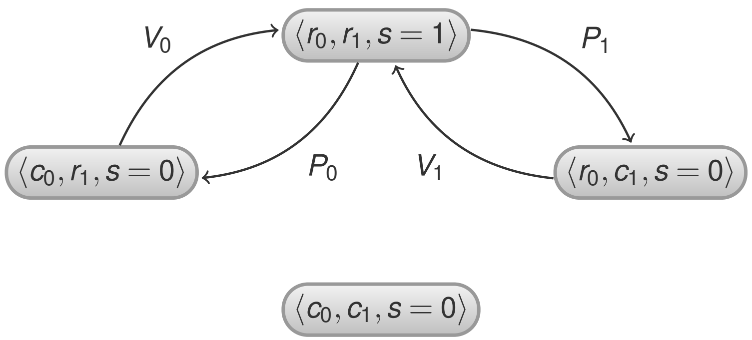
$$\Box(\mathit{proc}[0]@\mathit{Remainder} \rightarrow \Diamond \mathit{proc}[0]@\mathit{Critical})$$

does not.

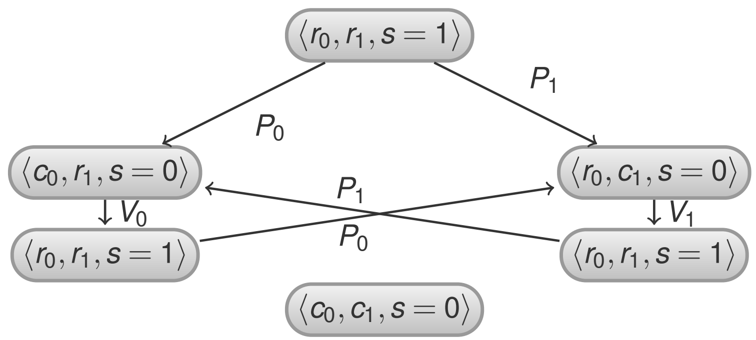
What is the problem?

- We have declared the meaning of a semaphore abstractly, without thinking about *fairness*
- SPIN is using exactly what we wrote and finds an unfair cycle
- We would like to prove properties about algorithms using semaphores *without* thinking about how the semaphores are implemented
- We would like to make fairness assumptions instead

Fair and unfair cycles



Fair and unfair cycles



Fairness

- Assumption that some part of a transition system eventually progresses, without quantitative restrictions
- Can be viewed as an abstraction of many possible transition scheduling policies
- There are many different notions of fairness
  - The most common is *weak fairness*
  - Another not uncommon one is *strong fairness*

Definition of fairness

Definition

Let A be a set of *actions* (or transitions). A computation $s_0 s_1 s_2 \cdots$ is

unconditionally fair with respect to A, if: When for every i we find an $j > i$ where an action in A is enabled in the state s_j, then we find also a $k > i$ where an action in A is performed from s_k.

weakly fair with respect to A, if: When for every i an action in A is enabled in the state s_j for every $j > i$, then we find also a $k > i$ where an action in A is performed from s_k.

strongly fair with respect to A, if: When an action in A is enabled infinitely often, then it is performed infinitely often.

Fairness assumptions in LTL

unconditional fairness $\Box\Diamond\Psi$

weak fairness $\Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$

strong fairness $\Box\Diamond\Phi \rightarrow \Box\Diamond\Psi$

Definition

A fairness assumption is a conjunction of unconditional fairness assumptions, weak fairness assumptions and strong fairness assumptions, i.e.:

$$fair \triangleq \left(\bigwedge_{i\in U} \Box\Diamond\Psi_i\right) \wedge \left(\bigwedge_{i\in W} \Diamond\Box\Phi_i \rightarrow \Box\Diamond\Psi_i\right) \wedge \left(\bigwedge_{i\in S} \Box\Diamond\Phi_i \rightarrow \Box\Diamond\Psi_i\right)$$

Fair satisfaction

Definition

We say that $M \models_{fair} \varphi$, if for all paths π in M with $\pi \models fair$ we have $\pi \models \varphi$.

Example

For the semaphore problem, define

$$fair \triangleq (\Box\Diamond P[0]@Remainder \rightarrow \Box\Diamond P[0]@Critical) \wedge$$
$$(\Box\Diamond P[1]@Remainder \rightarrow \Box\Diamond P[1]@Critical)$$

Remark

For the semaphore example, absence of starvation is now a trivial property.

Checking fairness in exploration algorithm

To verify φ under fairness assumption *fair*:

- Naive algorithm searches for bad loops π that satisfy

$$\pi \models \mathit{fair} \wedge \neg\varphi$$

- A more efficient solution for *weak fairness:*
  - search for bad loops that satisfy $\neg\varphi$ in which each action A with weak fairness is once either disabled or taken.

Enforcing fairness constraints

- Fairness can be expressed in LTL, but this may be difficult, inconvenient, or infeasible
- SPIN provides options to enforce default types of process scheduling fairness
- There is a cost associated with the implementation as part of the analysis algorithm:
  - weak fairness: linear increase of complexity (in number of processes)
  - strong fairness: quadratic increase of complexity (in number of processes)

The basic idea: unfolding

Flag construction method by Yaacov Choueka

- Create $(k+2)$ copies of the global reachability graph, with k the number of active processes, numbered from 0 to $k+1$.
- Preserve accept state labels only in the copy numbered 0
- change the transition relation to connect all $k+2$ copies:
  - in copy 0, change the destination state for outgoing transitions of all *accepting* states so that they point to their successors in copy 1
  - in copy $k+1$, change the destination state for all transitions so that they point to their successors in copy 0
  - in copy i for $1 \leq i \leq k$ change the destination state for all transitions contributed by process i to the corresponding state in copy $i+1$
  - add a *nil*-transition from any state in copy i where process i is blocked (has no enabled transitions) to the same state in copy $i+1$
- an accepting ω-run in the unfolded automation necessarily contains transitions from all active processes and therefore is weakly fair

Fair reminders

- SPIN's built-in notion of fairness applies only to
  - weak fairness, not strong fairness
  - process scheduling
  - not to non-deterministic choices within a process
- Other types of fairness can be expressed in LTL

Suggested reading

- (Baier & Katoen, 2008) 3.5

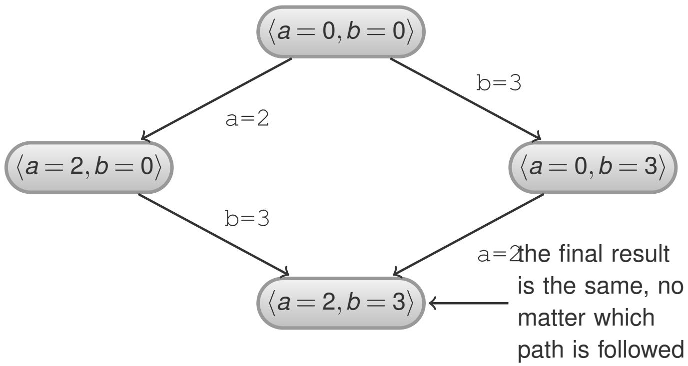
Partial order reduction

Partial order reduction

- Partial order reduction is a method to reduce the number of transitions taken during the search for counter examples,
- Idea: full asynchronous interleaving of process actions is sometimes redundant.

```
byte a, b;
active proctype P()
{
  a = 2; 0
}

active proctype Q()
{
  b = 3; 0
}
```

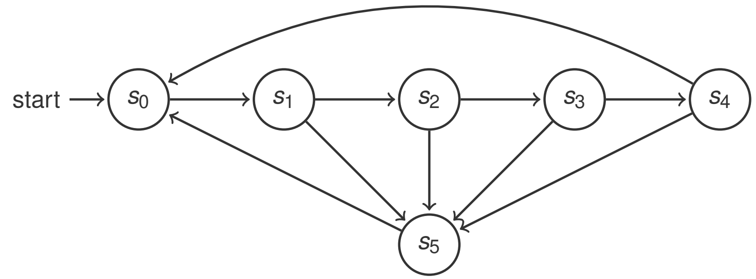


Computation tree logic

Computation tree logic

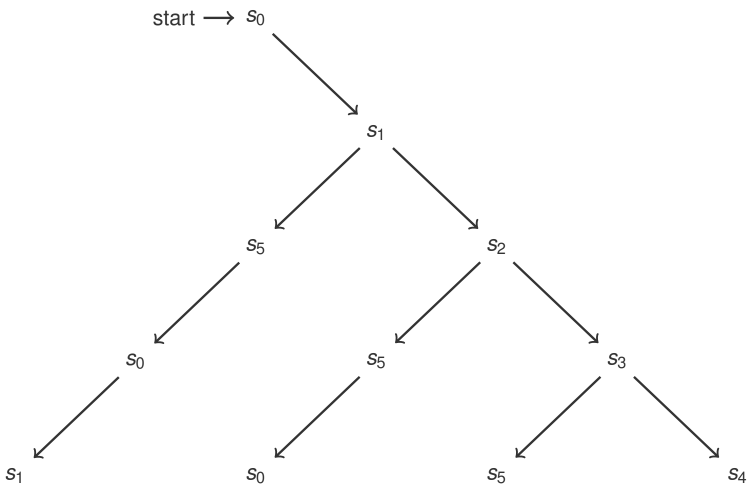
- Until today, we have been using LTL, a linear time logic. A linear time logic evaluates all properties on linear futures, which we called paths.
- But sometimes, we want to explore possibilities in the system. Consider the requirement “at any time we want to be able to abort”.
- Can we express this property in LTL?

Lets look at a transition system with this property



- We cannot say $\Box(\neg s_5 \rightarrow \bigcirc s_5)$, because this requires to always visit s_5 next.
- We actually want to say “it is always possible to visit s_5”, not that we will always do so!

Computation trees



Path quantifiers

- Solution: introduce *path quantifiers*
- $\exists\Diamond p$ means “there exists a path on which eventually p holds”
- $\forall\Box p$ means “on every path always p holds”
- Observe: in these examples, a *path quantifier* is always paired with a *temporal modality*. This is a defining feature of CTL (computation tree logic)

Syntax

Definition

A *state formula* over a set of atomic propositions P is formed by the following grammar:

$$\Phi ::= \mathit{true} \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists\Psi \mid \forall\Psi$$

where Ψ is a *path formula*. A CTL path formula is formed by the following grammar:

$$\Psi ::= \bigcirc\Phi \mid \Phi\,\mathcal{U}\,\Phi$$

where Φ represents a state formula.

Telematics Computer Systems
Symbolic model checking
Computation Tree Logic
Freie Universität Berlin

Abbreviations

- The Boolean connectives $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined in the usual way.
- The modalities for always and eventually can be derived in a similar way to LTL:
  - $\forall\Diamond\varphi \equiv \forall(\mathit{true}\,\mathcal{U}\,\varphi)$
  - $\exists\Diamond\varphi \equiv \exists(\mathit{true}\,\mathcal{U}\,\varphi)$
  - $\forall\Box\varphi \equiv \neg\exists\Diamond\neg\varphi$
  - $\exists\Box\varphi \equiv \neg\forall\Diamond\neg\varphi$

Telematics Computer Systems
Symbolic model checking
Computation Tree Logic
Freie Universität Berlin

Verbalisations

- $\exists\Diamond\varphi$ is pronounced "potentially φ"
- $\forall\Diamond\varphi$ is pronounced "inevitably φ"
- $\exists\Box\varphi$ is pronounced "potentially always φ"
- $\forall\Box\varphi$ is pronounced "invariantly φ"

Telematics Computer Systems
Symbolic model checking
Computation Tree Logic
Freie Universität Berlin

Semantics

Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system without terminal states, let state $s \in S$, φ, ψ CTL state formulae, and ξ be a path formula. Let *Paths*(s) be the set of all infinite execution fragments of T that start in s.

- $s \models a$ if and only if $a \in \mu(s)$
- $s \models \neg\varphi$ if and only if $s \not\models \varphi$
- $s \models \varphi \wedge \psi$ if and only if $s \models \varphi$ and $s \models \psi$
- $s \models \exists\xi$ if and only if for some $\pi \in \mathit{Paths}(s)$, $\pi \models \xi$.
- $s \models \forall\xi$ if and only if for every $\pi \in \mathit{Paths}(s)$, $\pi \models \xi$.
- $\pi \models \bigcirc\varphi$ if and only if $\pi(1) \models \varphi$
- $\pi \models \varphi\,\mathcal{U}\,\psi$ if and only if there exists j such that $\pi(j) \models \psi$ and for all $i < j$: $\pi(i) \models \varphi$.

where for a path $\pi = s_0 s_1 s_2 \cdots$ and natural number i $\pi(i) \triangleq s_i$ represents the $(i+1)$th state.

Semantics on transition systems

Definition

Given a transition system T, the satisfaction set $SAT_T(\varphi)$ for a CTL formula φ is defined by:

$$SAT_T(\varphi) = \{s \in S \mid s \models \varphi\}.$$

The transition system TS satisfies a CTL formula φ if and only if φ holds in all initial states of T:

$$T \models \varphi \text{ if and only if } \forall s_0 \in I : s_0 \models \varphi$$

This is equivalent to: $I \subseteq SAT_T(\varphi)$

We use an analogous definition for LTL.

Interpretation of several CTL formulae I

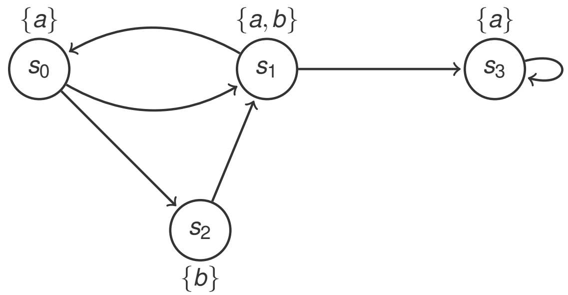


Figure: A transition system

Interpretation of several CTL formulae II

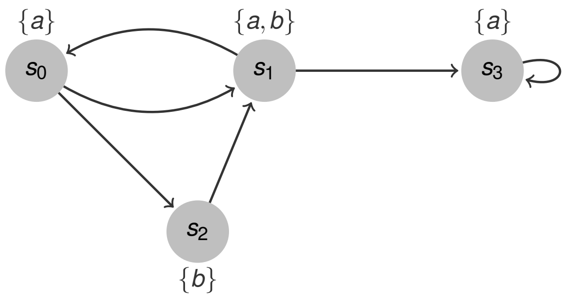


Figure: States in which $\exists \bigcirc a$ holds

Interpretation of several CTL formulae III

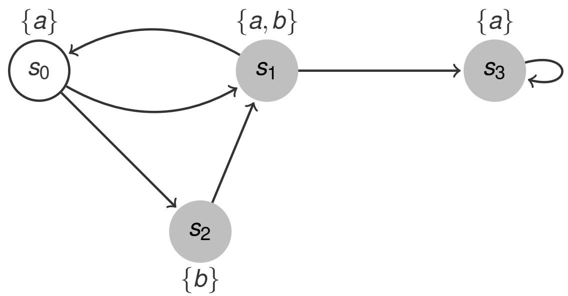


Figure: States in which $\forall \bigcirc a$ holds

Interpretation of several CTL formulae IV

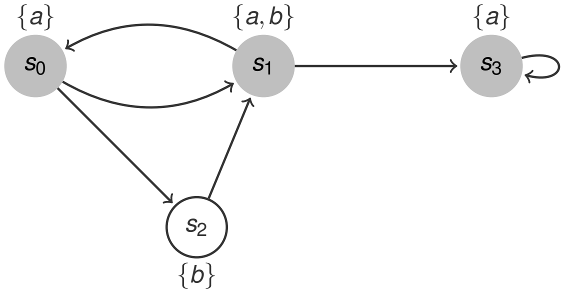


Figure: States in which $\exists\Box a$ holds

Interpretation of several CTL formulae V

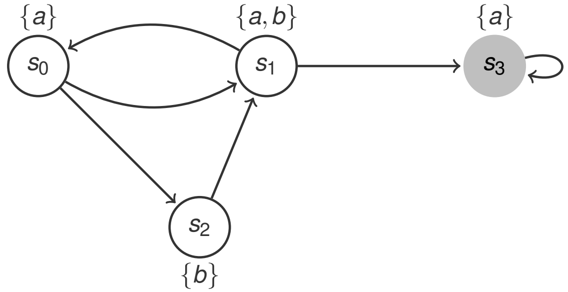


Figure: States in which $\forall\Box a$ holds

Interpretation of several CTL formulae VI

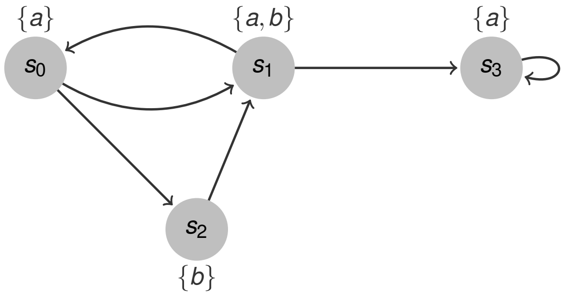


Figure: States in which $\exists\Diamond(\exists\Box a)$ holds

Interpretation of several CTL formulae VII

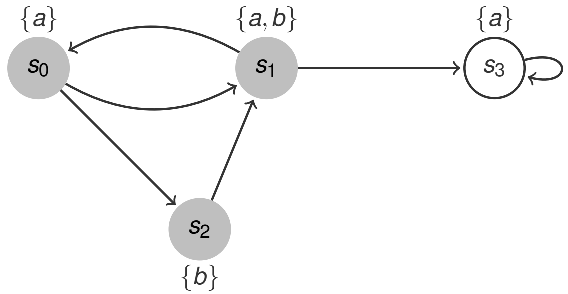


Figure: States in which $\forall(a\,\mathcal{U}\,b)$ holds

Interpretation of several CTL formulae VIII

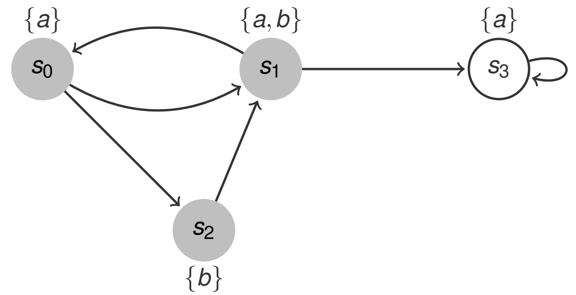


Figure: States in which $\exists(a\mathcal{U}\neg a \wedge \forall(\neg a \mathcal{U} b))$ holds

Useful equivalences for CTL I

Duality laws

$$\begin{aligned}
\forall \bigcirc \varphi &\equiv \neg\exists \bigcirc \neg\varphi \\
\exists \bigcirc \varphi &\equiv \neg\forall \bigcirc \neg\varphi \\
\forall \Box \varphi &\equiv \neg\exists\Diamond\neg\varphi \\
\exists \Box \varphi &\equiv \neg\forall\Diamond\neg\varphi \\
\forall(\varphi \mathcal{U} \psi) &\equiv \neg\exists(\neg\psi \mathcal{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\exists\Box\neg\psi \\
&\equiv \neg\exists((\varphi \wedge \neg\psi)\mathcal{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\exists\Box(\varphi \wedge \neg\psi)
\end{aligned}$$

Distributive laws

$$\begin{aligned}
\forall\Box(\varphi \wedge \psi) &\equiv \forall\Box\varphi \wedge \forall\Box\psi \\
\exists\Diamond(\varphi \vee \psi) &\equiv \exists\Diamond\varphi \vee \exists\Diamond\psi
\end{aligned}$$

Useful equivalences for CTL II

Expansion laws

$$\begin{aligned}
\forall(\varphi \mathcal{U} \psi) &\equiv \psi \vee (\varphi \wedge \forall\bigcirc\forall(\varphi\mathcal{U}\psi)) \\
\forall\Diamond\varphi &\equiv \varphi \vee \forall\bigcirc\forall\Diamond\varphi \\
\forall\Box\varphi &\equiv \varphi \vee \forall\bigcirc\forall\Box\varphi
\end{aligned}$$

$$\begin{aligned}
\exists(\varphi \mathcal{U} \psi) &\equiv \psi \vee (\varphi \wedge \exists\bigcirc\exists(\varphi\mathcal{U}\psi)) \\
\exists\Diamond\varphi &\equiv \varphi \vee \exists\bigcirc\exists\Diamond\varphi \\
\exists\Box\varphi &\equiv \varphi \vee \exists\bigcirc\exists\Box\varphi
\end{aligned}$$

Expressiveness of CTL vs. LTL

To keep a long story short: The expressiveness of CTL and LTL are incomparable. There are properties we can express in CTL which we cannot express in LTL and vice versa.
It is additionally worth-while to study the differences in expressive power.

Definition

A CTL formula φ is equivalent to a LTL formula ψ (both over P) if and only if for all transition systems T over P:

$$T \models \varphi \text{ if and only if } T \models \psi$$

We write $\varphi \equiv \psi$ if the formulae are equivalent.

Characterisation theorem

Theorem

Let φ be a CTL formula and ψ the LTL formula that is obtained from φ by eliminating all quantifiers. Then:

$$\varphi \equiv \psi \text{ or there does not exist any LTL formula that is equivalent to } \varphi$$

For a proof, see (Clarke & Draghicescu, 1989).

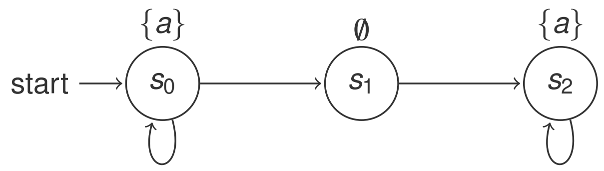
Persistence

Lemma

The CTL formula $\forall\Diamond\forall\Box a$ and the LTL formula $\Diamond\Box a$ are not equivalent.

Proof.

Consider the transition system below:



$\square$

Proof continued

The initial state s_0 clearly satisfies $\Diamond\Box a$, because every path starting in s_0 has the form $s_0^\omega + (s_0 * s_1 s_2^\omega)$, thus we stay forever in a state that satisfies a. Now look carefully at the path s_0^ω. This path does not satisfy $\forall\Diamond\forall\Box a$, because in this path it is always *possible* to take a transition to s_1, which does not satisfy a.

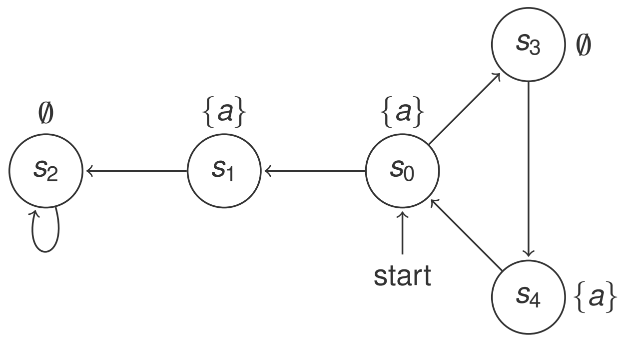
Corollary

$\Diamond\Box a$ is not expressible in CTL.

Eventually an *a*-State with only direct *a*-Successor

Lemma

The CTL formula $\forall\Diamond(a \wedge \forall\bigcirc a)$ and the LTL formula $\Diamond(a \wedge \bigcirc a)$ are not equivalent.



Proof.

Consider the previous figure. All paths have either $s_0 s_1$ or $s_0 s_3 s_4$ as a prefix. Clearly, all such paths satisfy the LTL formula $\Diamond(a \wedge \bigcirc a)$. On the other hand, $s_0 \not\models \forall\Diamond(a \wedge \forall \bigcirc a)$. Consider the path $s_0 s_1 s_2^\omega$. This one does not satisfy $\Diamond(a \wedge \forall \bigcirc a)$, because s_0 also has the successor s_3, which does not satisfy a. □

Incomparable expressiveness

Theorem

1. *There exist LTL formulae for which no equivalent CTL formula exists.*
2. *There exist CTL formulae for which no equivalent LTL formula exists.*

Example

1. There is no equivalent CTL formula for $\Diamond\Box a$ and $\Diamond(a \wedge \bigcirc a)$.
2. There is no equivalent LTL formula for $\forall\Diamond\forall$, $\forall\Diamond(a \wedge \forall \bigcirc a)$, and $\forall\Box\exists\Diamond a$.

CTL and fairness

- An important difference between LTL and CTL is, that we can express fairness assumptions in LTL, but we cannot express them in CTL.
- The observation is that $T \models_{fair} \varphi$ if and only if $T \models fair \rightarrow \varphi$
- This is not possible for CTL, because most fairness properties cannot be encoded in CTL.

Lemma

$\Box\Diamond b \rightarrow \Box\Diamond c$ is not expressible in CTL.

Proof.

We have $\Box\Diamond b \rightarrow \Box\Diamond c \equiv \Diamond\Box\neg b \vee \Box\Diamond c$ and the persistence property $\Diamond\Box\neg b$ cannot be expressed in CTL. □

Expressing fairness

Traditionally, one deals with fairness by *modifying* the semantics of transition systems to consider only the fair paths of a model, i.e. $T_{fair} = [\![T]\!] \cap [\![fair]\!]$, where *fair* is specified by LTL-like formulae.

Definition

A *compassion constraint* or *strong fairness constraint* is a set $\{(\varphi_i, \psi_i) \mid 1 \leq i \leq k\}$, where all φ_i, ψ_i are CTL formulae.

The compassion set is "interpreted" by the "special" formula

$$sfair = \bigwedge_{1 \leq i \leq k} (\Box\Diamond\varphi_i \rightarrow \Box\Diamond\psi_i)$$

(This specification uses the generalised logic $CTL^\star$)

Similar definitions can be given for weak fairness (*justice*) and unconditional fairness.

Model checking algorithm

- We do not work out algorithms for CTL model checking (maybe later)
- We also point out, that CTL model checking cannot be performed by SPIN
- An open source model checker for CTL is *NuSMV* (`http://nusmv.irst.itc.it/`)
- NuSMV uses a different input language for models.

The basic idea

for all $i \leq |\varphi|$
  for all $\psi \in \mathit{subformula}(\varphi)$ with $|\psi| = i$ **do**
    compute $\mathit{Sat}(\psi)$ from $\mathit{Sat}(\psi')$ **for**
      maximal proper $\psi' \in \mathit{subformula}(\psi)$
  return $I \subseteq \mathit{Sat}(\varphi)$

This traverses the formula to prove from the smallest subformulae upwards to the full construction, expanding or reducing the set of formulae satisfied at the current level.

Suggested reading

- (Baier & Katoen, 2008) 6.1–6.3

Symbolic Model Checking with Binary Decision Diagrams

Symbolic CTL Model Checking

- When model checking was introduced in 1981, memory and CPU was limited: the 80286 was not yet introduced, and servers had 4MB RAM.
- More powerful machines were not widely available, and were often too expensive for universities and research institutes.
- Thus, only small models could be verified at that time (up to 10^8 states in 1990s). Complexity results threatened to make model checking a curiosity (e.g., LTL model checking is P-SPACE hard).
- The landmark paper of Ken McMillan et.al., titled “Symbolic Model Checking: 10^{20} States and Beyond” (IEEE, 1990) introduced the idea of *symbolic model checking* using *reduced ordered binary decision diagrams.* (Bryant, 1986).
- These techniques work especially well with CTL.

A different view on transition systems

- In the sequel, $T = (S, \rightarrow, I, P, \mu)$ be a transition system (the transition labels are irrelevant and are omitted). Let $n \geq \max(\lceil \log |S| \rceil, 1)$ and choose an arbitrary, injective encoding $enc : S \rightarrow 2^n$ (where 2^n represents any set isomorphic to $\{0,1\}^n$).
- A set of states can be represented by its characteristic function $\chi : 2^n \rightarrow \{0,1\}$.
- The transition function can be encoded by a function $\Delta : 2^{2n} \rightarrow \{0,1\}$, where $\Delta(enc(s), enc(t)) = 1$ if and only if $s \rightarrow t$.

Switching functions I

- For technical reasons, it is more convenient to name the positions in the state vectors, making composition and analysis more simple.
- Let $z_1, \ldots, z_m$ be Boolean variables and $Var = \{z_1, \ldots, n_m\}$. Let $Eval(z_1, \ldots, z_m)$ denote the set of evaluations for $z_1, \ldots, z_m$, i.e. functions $\eta : Var \rightarrow \{0,1\}$. Evaluations are written as $\{z_1 \mapsto b_1, z_2 \mapsto b_2, \ldots, z_m \mapsto b_m\}$.
- A switching function for Var is a function $f : Eval(Var) \rightarrow \{0,1\}$. For $Var = \emptyset$, the switching functions are the constants 0 and 1.
- Disjunction, conjunction, and negation are defined in the obvious way:

$$
\begin{aligned}
(f \vee g)(\{z_1 \mapsto b_1, z_2 \mapsto b_2, \ldots, z_m \mapsto b_m\}) = \\
\max\{f(\{z_1 \mapsto b_1, z_2 \mapsto b_2, \ldots, z_m \mapsto b_m\}), \\
g(\{z_1 \mapsto b_1, z_2 \mapsto b_2, \ldots, z_m \mapsto b_m\})\}
\end{aligned}
$$

Switching functions II

- We write z_i for the *projection function* $pr_{z_i} : Eval(Var) \rightarrow \{0,1\}$ with $pr_{z_i}(\{\ldots, z_i \mapsto b_i, \ldots\}) = b_i$.
- With these notations, switching functions can be represented by boolean connections of the variables z_i, viewed as projection functions, and constants.
- $z1 \vee (z_2 \wedge \neg z_3)$ is a switching function.

Cofactor

Let $f : Eval(Var) \rightarrow \{0,1\}$ be a switching function.

- The *positive cofactor* of f is the switching function $f|_{z=1}(Var \setminus \{z \mapsto 0, z \mapsto 1\}) = f((Var \setminus \{z \mapsto 0, z \mapsto 1\}) \cup \{z \mapsto 1\})$
- The *negative cofactor* of f is the switching function $f|_{z=0}(Var \setminus \{z \mapsto 0, z \mapsto 1\}) = f((Var \setminus \{z \mapsto 0, z \mapsto 1\}) \cup \{z \mapsto 0\})$
- z is said to be *essential* for f if and only if $f|_{z=0} \neq f|_{z=1}$.

Example

Consider the switching function $f(z_1, z_2, z_3) = (z_1 \vee \neg z_2) \wedge z_3$. Then $f|_{z_1=0} = \neg z_2 \wedge z_3$ and $f|_{z_1=1} = z_3$. In particular, z_1 is essential for f.

Shannon expansion

Lemma

If f is a switching function for Var, then for each $z \in Var$:

$$f = f|_{z=0} \vee f|_{z=1}$$

Binary decision trees

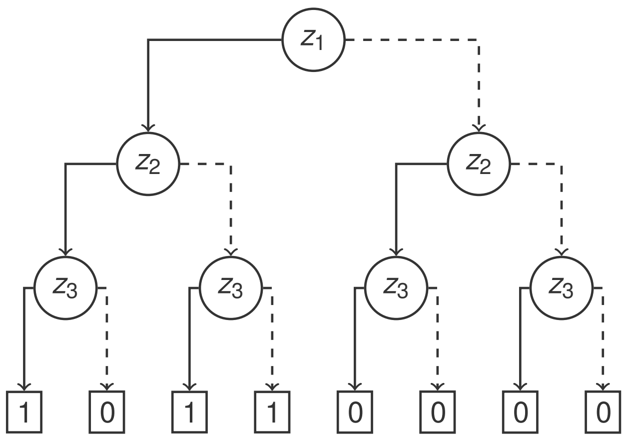


Figure: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$

Quantified Boolean formulae

Definition

$\exists z.f \triangleq f|_{z=0} \vee f|_{z=1}$

Definition

$\forall z.f \triangleq f|_{z=0} \wedge f|_{z=1}$

Renaming

Definition

Let *Var* be a set of variables and $f(Var)$ be a switching function. The switching function $f\{z \leftarrow y\}$ for $y \notin Var$ is the switching function over $Var \cup \{y\} \setminus \{z\}$ given by:

$$f\{z \leftarrow y\} = \begin{cases} f(s) & \text{if } s \neq y \\ f(z) & \text{if } s = y \end{cases}$$

Renaming can be generalised to sets of variable names in the natural manner.

Encoding transition systems by switching functions I

Given a transition system $T = (S, \rightarrow, I, P, \mu)$, we now build an equivalent representation as a switching function.

- For states S choose $k \triangleq \max\{\lceil \log_2 |S| \rceil, 1\}$ different Boolean variables $x_1, \ldots, x_k$ and define an arbitrary encoding function to map S into $Eval(x_1, \ldots, x_k)$.
- Subsets $B \subseteq S$ can be encoded by their *characteristic function*

$$\chi_B : Eval(x_1, \ldots, x_k) \rightarrow \{0,1\},\ \chi_B(s) = \begin{cases} 1 & \text{if } s \in B \\ 0 & \text{otherwise.} \end{cases}$$

- For any atomic proposition $a \in P$, the satisfaction set $Sat(a) = \{s \in S \mid a \in \mu(s)\}$ can be encoded by the switching function $f_a = \chi_{Sat(a)}$.

Encoding transition systems by switching functions II

- To encode the transition relation, we build a switching function over $\bar{x}, \bar{x}'$, where $\bar{x}'$ contains *primed* versions of members in $\bar{x}$.
- Intuition: $\bar{x}$ represents the current state, $\bar{x}$ represents a successor states.
- Then we identify $\rightarrow$ with the switching function

$$\Delta : Eval(\bar{x}, \bar{x}') \rightarrow \{0,1\},\ \Delta(s, t\{\bar{x}' \leftarrow \bar{x}\}) = \begin{cases} 1 & \text{if } s \rightarrow t \\ 0 & \text{otherwise} \end{cases}$$

Remark

Synchronous product becomes conjunction:
$\Delta(\bar{x}_1, \bar{x}_2, \bar{x}_1', \bar{x}_2') = \Delta_1(\bar{x}_1, \bar{x}_1') \wedge \Delta_2(\bar{x}_2, \bar{x}_2')$.
Asynchronous product becomes disjunction:
$\Delta(\bar{x}_1, \bar{x}_2, \bar{x}_1', \bar{x}_2') = \Delta_1(\bar{x}_1, \bar{x}_1') \vee \Delta_2(\bar{x}_2, \bar{x}_2')$.

What to do with it?

- Finally, the transition system has been encoded by a switching function.
- The model checking algorithm can now compute the satisfaction sets symbolically. Below is an example for computing $Sat(\exists(C \,\mathcal{U}\, B))$.

$f_0(\bar{x})$=$\chi_B(\bar{x})$;
j=0;
do {
  $f_{j+1}(\bar{x}) = f_j(\bar{x}) \vee (\chi_C(\bar{x}) \wedge \exists \bar{x}'.(\Delta(\bar{x}, \bar{x}') \wedge f_j(\bar{x}')))$;
  j++;
} **while** $(f_j(\bar{x}) \neq f_{j-1}(\bar{x}))$
return $f_j(\bar{x})$

Here, we have computed the *least fixed point* by doing a forward breadth-first search.

Finding a memory efficient representation

- The smart trick is to find a memory efficient representation for switching functions.
- Observation: Let m be a number of variables. Then there are at most 2^{2^m} different switching functions.
- We cannot expect to find a small representation for any possible switching function, but for the most common cases, we can.

Ordered binary decision diagrams

Idea: Skip redundant fragments of binary decision trees.

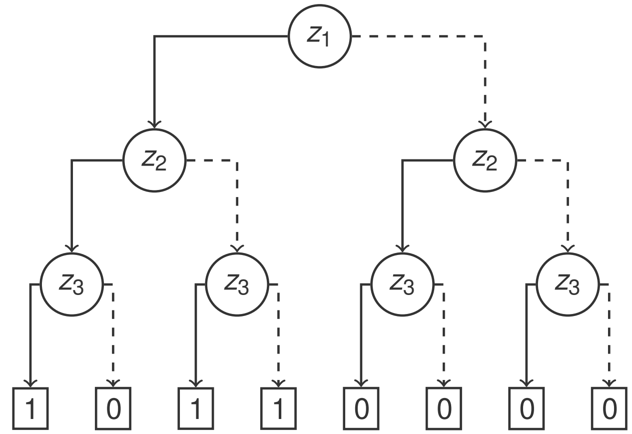


Figure: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$

. . . as a binary decision diagram

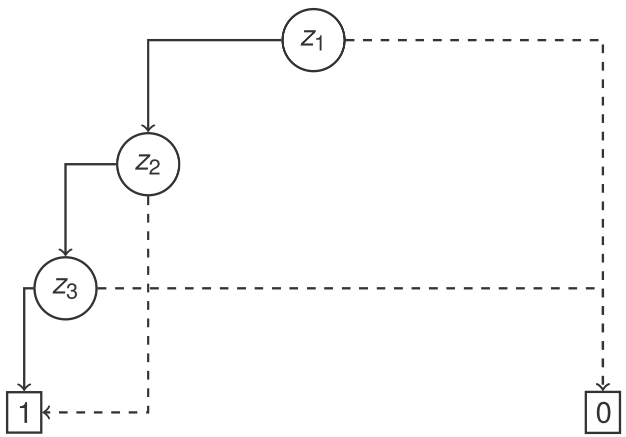


Figure: Binary decision tree diagram $z_1 \wedge (\neg z_2 \vee z_3)$

OBDD depend on variable order

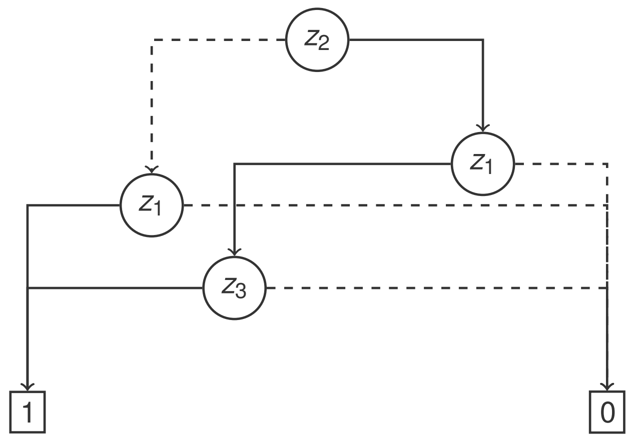


Figure: Binary decision tree diagram $z_1 \wedge (\neg z_2 \vee z_3)$

Definition

Let $\rho = (z_0, z_i, \ldots, z_k)$ be a variable ordering for *Var*. An ρ-ordered binary decision diagram (ρ-OBDD for short) is a tuple
$\mathfrak{B} = (V, V_I, V_t, succ_0, succ_1, var, val, v_0)$, where:

- V is a finite set of nodes, partitioned into V_I (inner nodes) and V_T (terminal nodes or *drains*.
- a root node $v_0 \in V$.
- successor functions $succ_{0,1} : V_I \rightarrow V \setminus \{v_0\}$ which are ρ-consistent, i.e.: $z_i = var(v) \wedge w \in \{succ_0(v), succ_1(v)\} \cap V_I \implies (z_j = var(w) \implies j > i$ and each node except root has a predecessor: $\forall v \in V \setminus \{v_0\} : \exists w \in V : \exists b \in \{0,1\} : v = succ_b(w)$
- a variable labelling function $var : V_I \rightarrow Var$
- a drain labelling function $val : V_T \rightarrow \{0,1\}$

Reduced ordered binary decision diagrams

- Let $\mathfrak{B}$ be a ρ-OBDD (where ρ represents the variable order). We call $\mathfrak{B}$ *reduced*, if for every pair $\{v, w\}$ of nodes in $\mathfrak{B}$: $v \neq w \implies f_v \neq f_w$. We write ρ-ROBDD for the reduced ρ-OBDD.

Canonicity

Theorem

Let Var be a finite set of boolean variables and ρ be a variable ordering for Var. Then:

- *For each switching function f for Var there exists a ρ-ROBDD $\mathfrak{B}$ with $f_{\mathfrak{B}} = f$.*
- *Given two ρ-ROBDD $\mathfrak{B}$ and $\mathfrak{C}$ with $f_{\mathfrak{B}} = f_{\mathfrak{C}}$, then $\mathfrak{B}$ and $\mathfrak{C}$ are isomorphic, i.e. agree up to renaming of the nodes.*

Corollary

Let $\mathfrak{B}$ be a ρ-ROBDD for f. Then $\mathfrak{B}$ is reduced if and only if $|\mathfrak{B}| \leq |\mathfrak{C}|$ for each ρ-ROBDD $\mathfrak{C}$ for f.

A switching function with small and exponential sized ROBDD

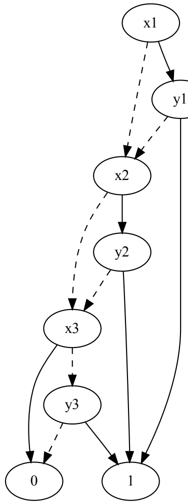
Example

Let $Var = \{x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m\}$ for $m \geq 1$ and

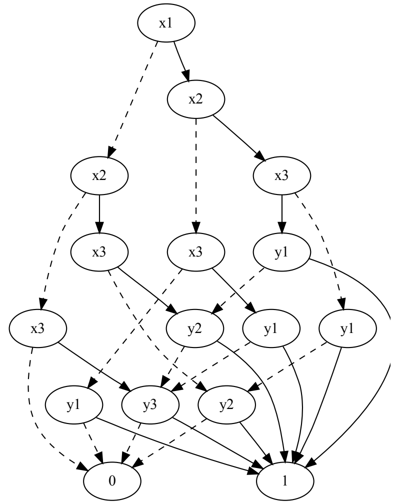
$$f_m = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_m \wedge y_m)$$

Then the $(x_m, y_m, x_{m-1}, y_{m-1}, \ldots, x_1, y_1)$-ROBDD has $2m + 2$ while $\Omega(2^m)$ nodes are required for the $(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m)$-ROBDD.

Small sized ROBDD



Exponential-sized ROBDD



Hardness of variable ordering

Lemma

Deciding, whether a variable ordering for a switching function is optimal (i.e., gives the least number of nodes) is NP-hard.

Theorem

Finding the optimal variable ordering for a switching function is NP-complete.

ROBDD are useful

- ROBDD are used ion many other places beyond (symbolic) model checking
- Their origin is *computer aided design* to synthesise circuits (logic synthesis)
- They are used in fault tree analysis
- They are used in Bayesian reasoning (e.g., spam filters)
- They are used in data flow analysis
- They are used for constraint solving (e.g., data bases)
- ...

Example implementations

- BuDDy (`http://sourceforge.net/projects/buddy/`)
- JINC (`http://www.jossowski.de/projects/jinc/jinc.html`)
- CUDD (`http://vlsi.colorado.edu/~fabio/CUDD/`)
- JDD (`http://javaddlib.sourceforge.net/jdd/`)
- ...

Complexity results

Theorem

For a transition system T with N states and K transitions, CTL formula φ and CTL fairness assumption fair with k conjuncts, the CTL model-checking problem $T \models_{fair} \varphi$ can be decided in time $O((N+K) \cdot |\varphi| \cdot k)$.

We did not discuss the problem, but state the theorem here:

Theorem

For a transition system T with N states and K transitions, CTL formula φ and CTL fairness assumption fair with k conjuncts, the CTL counter example can be generated in time $O((N+K) \cdot k)$.

CTL* is an extension of CTL

Definition

A *CTL* state formula* over a set of atomic propositions P is formed by the following grammar:

$$\Phi ::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists\Psi$$

where Ψ is a *CTL* path formula*. A CTL* path formula is formed by the following grammar:

$$\Psi ::= \Phi \mid \Psi \wedge \Psi \mid \neg\Psi \mid \bigcirc\Psi \mid \Psi \mathcal{U} \Psi$$

where Φ represents a state formula.

Semantics I

Let $T = (S, A, \rightarrow, I, P, \mu)$ be a transition system without terminal states, let state $s \in S$, $\varphi, \varphi_1, \varphi_2$ CTL* state formulae, and ψ, ψ_1, ψ_2 be CTL* path formulae.

- $s \models a$ if and only if $a \in \mu(s)$
- $s \models \neg\varphi$ if and only if $s \not\models \varphi$
- $s \models \varphi_1 \wedge \varphi_2$ if and only if $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \exists\psi$ if and only if for some $\pi \in Paths(s)$, $\pi \models \psi$.

Semantics II

- $\pi \models \varphi$ if and only if $\pi(0) \models \varphi$
- $\pi \models \neg\psi$ if and only if $\pi \not\models \psi$
- $\pi \models \psi_1 \wedge \psi_2$ if and only if $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models \bigcirc\psi$ if and only if $\pi^1 \models \psi$
- $\pi \models \psi_1 \mathcal{U} \psi_2$ if and only if there exists j such that $\pi^j \models \psi_2$ and for all $i < j$: $\pi^i \models \psi_1$

Remark

$\exists\varphi \equiv \varphi$ and $\forall\varphi \equiv \varphi$

Semantics on transition systems

Definition

Given a transition system T, the satisfaction set $SAT_T(\varphi)$ for a CTL* formula φ is defined by:

$$SAT_T(\varphi) = \{s \in S \mid s \models \varphi\}.$$

The transition system TS satisfies a CTL formula φ if and only if φ holds in all initial states of T:

$$T \models \varphi \text{ if and only if } \forall s_0 \in I : s_0 \models \varphi$$

This is equivalent to: $I \subseteq SAT_T(\varphi)$

Embedding CTL and LTL into CTL*

Lemma

For every transition system T without a terminal state, and every state s of T and every CTL formula ψ:

$$s \models^{CTL} \psi \text{ if and only if } s \models^{CTL*} \psi$$

Theorem

For every transition system T without a terminal state, and every state s of T and every LTL formula ψ:

$$s \models^{LTL} \psi \text{ if and only if } s \models^{CTL*} \forall\psi$$

Expressive power

Theorem

There does not exist any equivalent LTL or CTL formula for the CTL formula*

$$(\forall\Diamond\Box a) \vee (\forall\Box\exists\Diamond b)$$

over $P = \{a, b\}$

Summary

- Computation tree logic (CTL) is a logic for formalising properties over computation trees, i.e. branching structures
- Linear time logic is a logic for formalising properties about computation paths
- The expressiveness of LTL and CTL are incomparable
- Though fairness constraints cannot be encoded in CTL, fairness assumptions can be incorporated in CTL by adapting the CTL semantics such that quantification is over fair paths, rather than over all paths
- The CTL model-checking problem can be solved by a recursive descent procedure over the parse tree of the formula to be checked.
- Counter examples and witnesses for CTL can be determined using a standard graph analysis.

Suggested reading

- (Baier & Katoen, 2008) 6.7

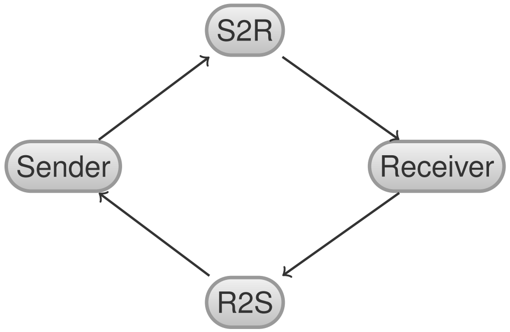
Case study: Alternating Bit Protocol

Let’s look at a case study

- We have learnt about two major techniques for verifying protocols and establish program correctness.
- It is useful to compare these two methods, because the tools perform differently.
- An open problem is: When is it better to prefer symbolic model checking over explicit state model checking.
- Once You have analysed a couple of models using the different tools, you will gain insight and intuition, which will enable you to make an informed choice.

The alternating bit protocol

The alternating bit protocol is *the* inevitable example in model checking courses.
The alternating bit protocol shall ensure that incoming data is delivered, but also in the right order.



Context of the protocol

- Sending messages over unreliable channels:
  - Messages may be lost
  - Messages may be distorted
  - Messages may be sent in different order
- Idea: Sender appends an alternating bit to each message
- Receiver checks that bits and confirms that it received the right message or asks for retransmission.

A Promela model I

```
mtype = { msg0, msg1, ack0, ack1 };

chan    to_sndr = [2] of { mtype };
chan    to_rcvr = [2] of { mtype };

active proctype Sender()
{
again:  to_rcvr!msg1;
        to_sndr?ack1;
        to_rcvr!msg0;
        to_sndr?ack0;
        goto again
}
```


A Promela model II

```
active proctype Receiver()
{
again:  to_rcvr?msg1;
        to_sndr!ack1;
        to_rcvr?msg0;
        to_sndr!ack0;
        goto again
}
```

ABP with losses I

```
active proctype Sender()
{
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit);
       r_s ? ack(seqno);
       if
       :: seqno == sbit ->
           sbit = 1 - sbit;
           data ++
       :: else
       fi
    od
}
```


ABP with losses II

```
active proctype Receiver()
{
    byte recd, expected = 0;
    bit rbit, seqno = 0;
    do
    :: s_r?msg(recd, seqno) ->
       if
       :: r_s!ack(seqno)
       :: skip
       fi;
       if
       :: seqno == rbit ->
           rbit = 1 - rbit;
       :: else
       fi
    od
}
```


ABP with losses and retransmission I

```
active proctype Sender()
{
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit);
    :: skip
    :: r_s ? ack(seqno);
       if
       :: seqno == sbit ->
           sbit = 1 - sbit;
           data ++
       :: else
       fi
    od
}
```


ABP with losses and retransmission II

```
active proctype Receiver()
{
    byte recd, expected = 0;
    bit rbit, seqno = 0;
    do
    :: s_r?msg(recd, seqno);
       if
       :: seqno == rbit ->
           rbit = 1 - rbit;
           assert(recd == expected);
           expected++
       :: else
       fi
    :: r_s!ack(rbit)
    :: skip
    od
}
```

ABP with message checking I

```
active proctype Sender()
{
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit);
    :: skip
    :: r_s ? ack(seqno);
       if
       :: seqno == sbit ->
           sbit = 1 - sbit;
           data ++
       :: else
       fi
    od
}
```


ABP with message checking II

```
active proctype Receiver()
{
    byte recd, expected = 0;
    bit rbit, seqno = 0;
    do
    :: s_r?msg(recd, seqno);
       if
       :: seqno == rbit ->
            rbit = 1 - rbit;
            assert(recd == expected);
            expected++
       :: else
       fi
    :: r_s!ack(rbit)
    :: skip
    od
}
```


ABP with progress I

```
active proctype Sender()
{
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: (data < 10) -> s_r ! msg(data, sbit);
    :: skip
    :: r_s ? ack(seqno);
       if
       :: seqno == sbit ->
           sbit = 1 - sbit;
           data ++
       :: else
       fi
    od
}
```


ABP with progress II

```
active proctype Receiver()
{
    byte recd, expected = 0;
    bit rbit, seqno = 0;
    do
    :: s_r?msg(recd, seqno);
       if
       :: seqno == rbit ->
            rbit = 1 - rbit;
            progress: assert(recd == expected);
            expected++
       :: else
       fi
    :: r_s!ack(rbit)
    :: (1) -> progress2: skip
    od
}
```

The ABP in NuSMV

- NuSMV does not have channels, so we have to formalise our own channels first. Message loss will be handled in the channel, which is a conceptual advantage.
- In NuSMV, the behaviour is formulated as switching functions, there is no state or location, only the values of the variables.

Channels

```
MODULE one-bit-chan(input)
  VAR
    output : boolean;
  ASSIGN
    next(output) := {input, output};
  FAIRNESS running

MODULE two-bit-chan(input1, input2)
  VAR
    output1 : boolean;
    output2 : boolean;
  ASSIGN
    next(output2) := {input2, output2};
    next(output1) := case input2 = next(output2) : input1 ;
                          1 : {input1, output1};
                     esac ;
  FAIRNESS running
```


Sender

```
MODULE sender(ack)
VAR
  st : {sending, sent};
  message1: boolean;
  message2: boolean;
ASSIGN
  init(st) := sending;
  next(st) := case ack = message2 & !(st = sent) : sent;
                 1 : sending;
              esac;
  next(message1) := case st = sent : {0, 1};
                        1 : message1;
                    esac;
  next(message2) := case st = sent : !message2;
                        1: message2;
                    esac;
  FAIRNESS running
```


Receiver

```
MODULE receiver(message1, message2)
VAR
  st : {receiving, received};
  ack : boolean;
  expected : boolean;
ASSIGN
  init(st) := receiving;
  next(st) := case message2 = expected & !(st = received): received;
                 1 : receiving;
              esac;
  next(ack) := case st = received : message2;
                  1 : ack;
               esac;
  next(expected) := case st = received : !expected;
                       1 : expectec;
                    esac;
  FAIRNESS running
```

Wiring everything together

```
MODULE main
VAR
  S : process sender(ack_chan.output);
  R : process receiver(msg_chan.output1, msg_chan.output2);
  msg_chan : process two-bit-chan(S.message1, S.message2);
  ack_chan : process one-bit-chan(R.ack);

ASSIGN
  init(S.message2) := 0;
  init(R.expected) := 0;
  init(R.ack) := 1;
  init(msg_chan.output2) := 1;
  init(ack_chan.output) := 1;
```


Summary

- The alternating bit protocol can be modelled in both languages with reasonable effort.
- Specifications that express the same property can usually be defined in both formalism; CTL specs tend to be weaker, however.
- The performance on the ABP is similar; our computers got to fast to let us see a difference.
- It is not obvious how we should compare both models; they are different implementations of the protocol and make different assumptions.

Formal abstractions

Formal abstractions

- Sometimes, the model checking tools can apply algorithms successfully to reduce the state space to search.
- Often, this state space is still to large for effective verification.
- We discuss methods for reducing the state space by using our brains and establishing the required properties of the model.
- The basic idea is to build a reduced model of the model or system to verify, verify the reduced model and *prove* that the verification result can be transferred to the bigger one.

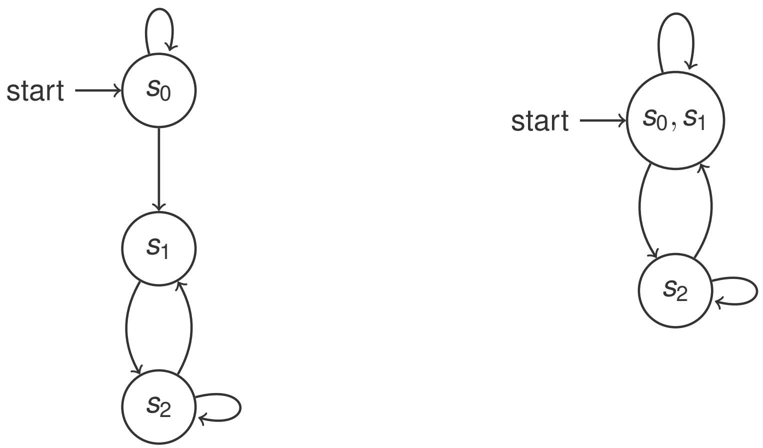
Linear-time branching-time spectrum

- What shall be the intuition of “bigger model” and “smaller model”?
- We shall define a *partial order* on transition systems, that effectively tells us which one is bigger...
- The guiding principle: a model is at least as big as another one, if it can be used instead of the other (substitution principle)
- This means that the bigger model has less behaviour (but more states)

Trace inclusion

- A (very weak) relation between two transition systems is given by *trace inclusion:*
- Let $\mathfrak{A}, \mathfrak{C}$ be two transition systems such that $[\![\mathfrak{C}]\!] \subseteq [\![\mathfrak{A}]\!]$.
- Then every universal property satisfied by $\mathfrak{A}$ is also satisfied by $\mathfrak{C}$.
- It might be nice, if $|\mathfrak{A}| \leq |\mathfrak{C}|$.
- The trick is to merge states but preserve outgoing transitions.

Example



What kind of properties are preserved?

- The important question for model checking is: what kind of properties are preserved?
- We desire: $\mathfrak{A} \models \varphi$ implies $\mathfrak{C} \models \varphi$ (soundness) and $\mathfrak{A} \not\models \varphi$ implies $\mathfrak{C} \not\models \varphi$ (completeness).
- There are little techniques which give us both.

Formalisation

Let $\mathfrak{C} = (S_C, \rightarrow_C, I_C, P, \mu_C)$ be a transition system which we call the *concrete* one. Let $\mathfrak{A} = (S_A, \rightarrow_A, I_A, P, \mu_A)$ be a transition system which we call the *abstract* one.

A binary relation $R \subseteq S_C \times S_A$ is called a *simulation relation*, if and only if:

1. For every $s_C \in I_C$ there exists $s_A \in I_A$ with $(s_C, s_A) \in R$.
2. For every pair $(s_C, s_A) \in R$, then for all s'_C with $s_C \rightarrow_C s'_C$ there exists s'_A with $s_A \rightarrow_A s'_A$ and $(s'_C, s'_A) \in R$.
3. It holds $\mu_C(s_C) = \mu_A(s_A)$ for all $(s_C, s_A) \in R$.

If there exists such an R, we say that $\mathfrak{A}$ simulates $\mathfrak{C}$ and write $\mathfrak{C} \sqsubseteq_R \mathfrak{A}$.

Under-approximation

If $\mathfrak{C} \sqsubseteq \mathfrak{A}$, then we say that $\mathfrak{C}$ is an *under-approximation* of $\mathfrak{A}$.

Lemma

When $\mathfrak{C} \sqsubseteq \mathfrak{A}$ *and* φ *is an* $\exists$-CTL* *property, then* $\mathfrak{C} \models \varphi$ *implies* $\mathfrak{A} \models \varphi$.

Over-approximation

If $\mathfrak{C} \sqsubseteq \mathfrak{A}$, then we say that $\mathfrak{A}$ is an *over-approximation* of $\mathfrak{C}$.

Lemma

When $\mathfrak{C} \sqsubseteq \mathfrak{A}$ *and* φ *is an* $\forall$-CTL* *property, then* $\mathfrak{A} \models \varphi$ *implies* $\mathfrak{C} \models \varphi$.

Suggested reading

- Principles of Model Checking (Baier & Katoen, 2008), Sections 7.4–7.5
- (McMillan, 2000)
- (Clarke, Long, & McMillan, 1989)
- (ACM, 1989)
- (Wolper & Lovinfosse, 1989)

Case Study:

Futurebus+ Cache Coherence Protocol

Verifying the Futurebus+ Cache Coherence Protocol

- The Futurebus+ is the specification of a computer bus (like ISA, PCI, PCI-E)
- It was intended to be *the* bus architecture for computers, connecting CPU, memory, and serving to some extend as LAN
- The protocol is described in (IEEE, 1994) using timing diagrams and predicate logic formulae.

Related work

- Analysis using model checking reported in (Clarke et al., 1993). Several mistakes were found and corrected in the standard.
- Uniform proof reported in (Kyas, 2001).

History and Background

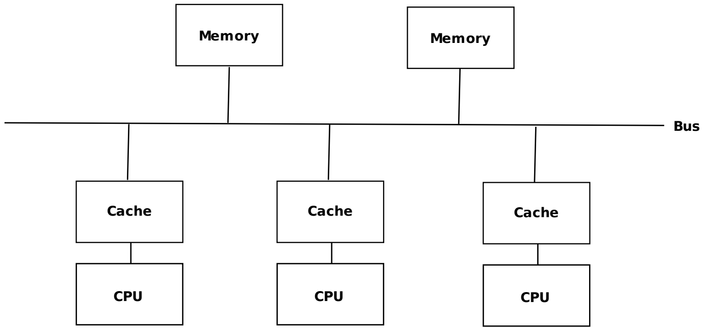
1979 A standardisation effort for the computer bus of the future was initiated, when devices plugged into existing busses became faster than the provided bandwidth

1987 After years of development (no product was present), a first version of the Futurebus standard was published. The US NAVY promised to standardise on Futurebus after changes

1994 The Futurebus+ standard was finalised, supporting multiple *profiles* to satisfy the needs of all stake holders. Unfortunately, improvements to conventional bus systems like PCI made Futurebus+ obsolete.

Futurebus+ was never widely adopted. Much research on *cache coherence* was initiated as part of this standardisation process.

Multiprocessors, caches and memory



Cache coherence

- Integrity of data stored in local caches.
- CPU read common memory addresses and manipulate them independently, thus leading to inconsistent data
- Related to memory coherence and *memory consistence,* the guarantees made by the system on concurrent access to memory

Characterisation

1. A read made by a processor P to a location X that follows a write by the same processor P, with no writes to X by other processors occurring between these two events, must always return the value written by P.
2. A read made by P_1 to X that follows a write by P_2 to X must return the value written by P_2 if no other write to X occurred in between.
3. Writes to the same location must be sequenced, i.e., if values A and B are written to X in that order, no processor may read B and then A from X.

Cache line states

invalid The content of the cache line is not valid or not present

shared The content of the cache line is shared with other CPU

exclusive-unmodified The cache line is owned exclusively by the CPU and has the same value as in memory

exclusive-modified The cache line is owned exclusively by the CPU and has possibly a different value from the one in memory

Abbreviations

- *readable*, if the state is shared, exclusive-unmodified or exclusive modified.
- *writable*, if the state is exclusive-modified.
- *unmodified*, if the state is shared or exclusive-unmodified.
- *exclusive*, if the state is exclusive-unmodified or exclusive modified.

Looking at some code

```
CMD=none:
  case
  -- Shared copies can be nondeterministically kicked out of the cac
  -- provided we aren't waiting for them to become exclusive.
  state=shared:
    case
    requester=exclusive: shared;
    1: {invalid, shared};
    esac;
  -- Exclusive unmodified copies can be nondeterministically kicked
  -- or written to.
  state=exclusive-unmodified:
  case
    SR: {invalid, shared};
    1: {invalid, shared, exclusive-unmodified, exclusive-modified};
  esac;
  1: state;
  esac;
```


Formalising cache coherence in CTL

1. $\forall\Box(p_1.\mathit{writable} \implies \neg p_2.\mathit{readable}$
2. $\forall\Box(p_1.\mathit{readable} \wedge p_2.\mathit{readable} \implies p_1.\mathit{data} = p_2.\mathit{data}$
3. $\forall\Box(p.\mathit{readable} \wedge \neg m.\mathit{memory\_line\_modified} \implies p.\mathit{data} = m.\mathit{data}$
4. $\forall\Box\exists\Diamond p.\mathit{readable}$
5. $\forall\Box\exists\Diamond p.\mathit{writable}$

Protocol description

- Memory is partitioned into lines, usually multiple machine words, which form one unit of transaction.
- Commands on the bus announce the intention to read one line with a specific purpose or that the line has been modified

read-shared	read with intention to share access
read-modified	read with intention to modify
none	no command
wait	wait
invalidate	announce that the value has been changed
copyback	write the value back to memory

Commands and modifiers

1. One node is chosen to be the master for this round
2. The master decides who will issue a command
3. All other nodes may *modify* the command by raising flags:
   - *sr*: if the command is read-shared, announces that more than one processor reads
   - iv: indicate that the value in memory is invalid and provided by another cache
   - *tf*:

Abstractions

- Only one bit of data and one memory line is considered.
-

Roadmap

1. What are *parameterised networks*?
2. How to specify a parameterised network?
3. How to specify properties of parameterised networks independent of the number of components?
4. Computing an abstract system from a parameterised network.
5. Model checking.
6. Results.

Parameterised Networks

What are *parametrised networks?*

- infinite family of finite-state processes,
- composed by parallel composition operators,
- parameterised by the number of processes.

$$P_1 \| P_2 \| \cdots \| P_n \quad \text{with Parameter } n.$$

Problems

1. How to specify properties independently of the
   - number of processes?
   - structure of the network?
2. How to specify a parameterised network and its *topology*?

Here: *only* specification of properties, *only* linear parameterised networks.

Verification Problem

Wanted: *uniform* proof of correctness for

- a specification φ
- a parametrised network $\mathscr{F}$

decide:

$$\forall P.P \in \mathscr{F} \wedge (P \models \varphi)$$

Theorem (Apt & Kozen, 1986)

The verification problem is undecidable in general, even if it is decidable for each member of a family.

Roadmap (refined)

specific transition systems (BTS)

specific linear parameterised networks (SMPS)

$$\text{WS1S} \left\{ \begin{array}{c} \text{bisimilar transition system} \\ \Downarrow \alpha \\ \text{finite state system} \end{array} \right. \qquad \text{Abstraction}$$

Specification:

- Universal properties
- Invariance

Boolean Transition Systems

To define components of a parameterised network:

- Transition system $S(i,n)$ as a predicate,
- Parameters i (process identifier), n (number of processes),
- propositional variables $b[i]$ (b is boolean array).

Definition

The *synchronous monadic parameterised system* (SMPS) built from an BTS $S(i,n)$ is the set

$$\mathscr{P} = \{ ||_{\ell=0}^{m-1} S(\ell,m)[i/\ell, n/m][v/v(\ell)]_{v \in V} \mid m < \omega \} \quad .$$

Key Observations

- A state of a linear system configuration is a *string* of states of its constituents.
- collect *related* states into a *regular language*.
- Describe this language in *WS1S*.
- *Transitions* lead from regular language to regular language.
- Properties are described in WS1S *easily* and *independently* of the number of processes in P'.

Weak Second Order Theory of One Successor (WS1S)

- *Sets* represent *finite* subsets of ω
- *Monadic* relations: $x \in S$ ($x < y$ is a special case)
- only *successor* function $\mathrm{succ}(x)$
- Quantification over natural numbers, monadic relations
- Constants 0 and $\emptyset$

Lemma

WS1S is decidable (Büchi 1960, Elgot 1961)

Central observation in the proof

WS1S can be decided by *finite state automatons,* its generalisation S1S (a.k.a. MSO) can be decided by *Büchi-automatons*.

Tool for deciding WS1S: MONA (`http://www.brics.dk/mona/`)

WS1S transition systems

Definition

A WS1S-TS $S = (V, \Theta, \mathscr{T})$ consists of

- A finite set V of second-order variables.
- A satisfiable predicate Θ with the free variables ranging over V.
- A finite set $\mathscr{T}$ of transitions described by WS1S formulae $\rho(V, V')$ with its free variables ranging over V and V'.

Computations are defined as usual.

Procedure

1. Compute a WS1S-TS bisimilar to SMPS
2. Compute an *abstract finite-state* transition system from the WS1S-TS when an *abstraction relation* α is supplied.
3. use a model checker, e.g. NUSMV, to check the specification.

Converting a SMPS to a WS1S-TS

For a SMPS $\mathscr{P}$ build from BTS $S(i, n)$ let μ denote the substitution replacing

- all occurences of the form $b[i]$ by $i \in X_b$.
- all occurences of n by $\max(P) + 1$.

Algorithm

$\tilde{P} = (\tilde{V}, \tilde{\Theta}, \tilde{\mathscr{T}})$ is defined by:

$$\tilde{\Theta} = \exists n. P = \{0, \ldots, n-1\} \wedge \forall m. m \in P \rightarrow \Theta\mu \tag{1}$$

$$\tilde{\mathscr{T}} = \{\exists_{\tau \in \mathscr{T}} Y_\tau.\mathrm{part}(P, \{Y_\tau | \tau \in \mathscr{T}\}) \wedge P = P' \wedge (\forall_{\tau \in \mathscr{T}} m_\tau. \bigwedge_{\tau \in \mathscr{T}} m_\tau \in Y_\tau \rightarrow \rho_\tau(V, V')\mu)\} \tag{2}$$

$$\tilde{V} = FV(\tilde{\Theta}) \cup FV(\tilde{\mathscr{T}}) \tag{3}$$

where

$$\mathrm{part}(Y, \{Y_\tau | \tau \in \mathscr{T}\}) = (\bigwedge_{\{\tau,\tau'\} \subseteq \mathscr{T}} Y_\tau \cap Y_{\tau'} = \emptyset) \wedge (Y = \bigcup_{\tau \in \mathscr{T}} Y_\tau)$$

Abstract State Graph

Computing the abstract state graph with:

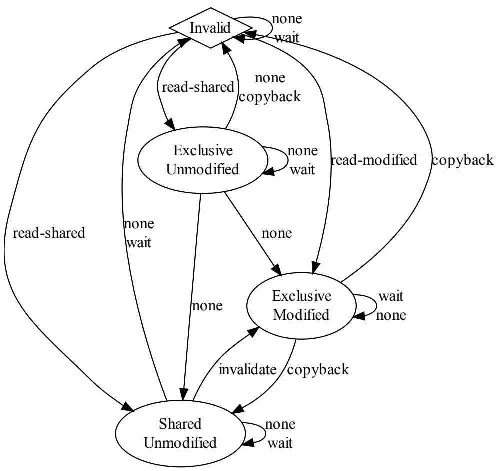
Theorem

For an L-simulation α (represented as a predicate) as a WS1S formula define

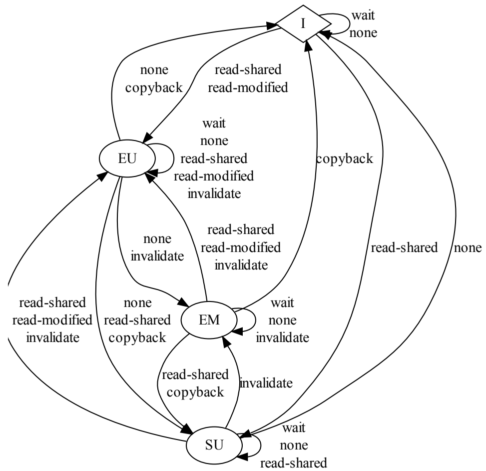
- $\Theta_\alpha = \{s \,|\, s \models \Theta \wedge \alpha\}$
- $\mathscr{T}_\alpha = \{(s, s') \,|\, (s, s') \models \alpha \wedge \rho \wedge \alpha'\}$

Then $S_\alpha = (FV(\alpha), \Theta_\alpha, \mathscr{T}_\alpha)$ is a finite state transition system.

Example: Futurebus+ Cache Coherence Protocol



Futurebus+ Cache Coherence Protocol (Abstract System)



Suggested reading

- (Clarke et al., 1993)
- (Kyas, 2001)
- (Clarke, Grumberg, & Peled, 1999)

Timed Automatons

Timed automatons

Today's material

Today's material is based on Chapter 9 of (Baier & Katoen, 2008) and Chapter 1 of (Olderog & Dierks, 2008).

–

Reactive systems

- Up to now, we have studied model checking of reactive systems, i.e. systems that maintain an ongoing interaction with an environment.

Real-time systems

– Very often, systems need to react in time, like traffic lights, air bags, and communication protocols

Correctness in time-critical systems not only depends on the logical result of the computation but also on the time at which the results are produced

Time domains

Fictitious time Time is recorded by a step counter, i.e. time is a natural number, during which things happen. If fictitious time increases by 1, we do not know how much it increased in reality. It allows to reason about *simultaneous* events

Discrete time Time is recorded by a step counter, i.e. time is a natural number, regardless of what is happening. Discrete time is used on hardware and 1 time unit often represents one clock cycle.

Dense time Time is "measured" in a physical unit, but recorded in a dense domain, e.g. non-negative rational numbers. This domain is usually used in tools.

Continuous time Time is "measured" in a physical unit and the domain is continuous, e.g. non-negative real numbers.

Introductory example

Figure: Railroad crossing

Modelling the train

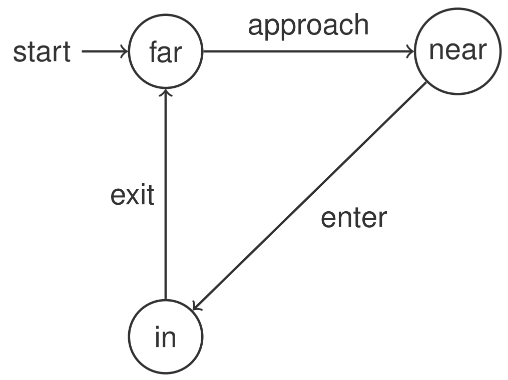


Figure: Train model

Modelling the controller

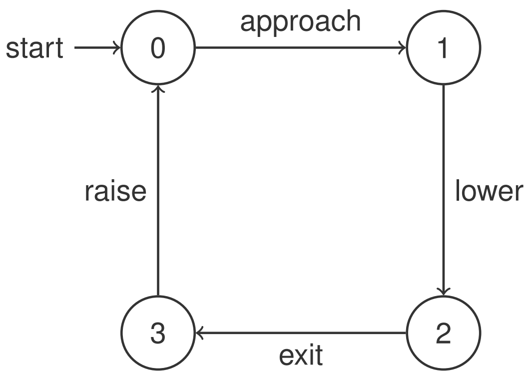


Figure: Controller model

Modelling the gate

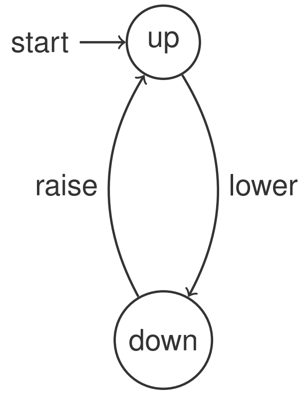


Figure: Gate model

Initial fragment of *Train* || *Controller* || *Gate*

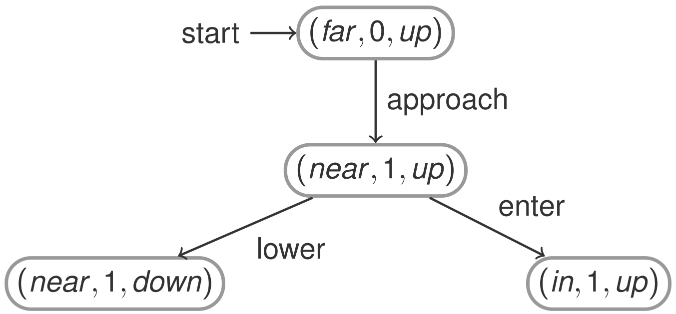


Figure: Initial fragment of *Train* || *Controller* || *Gate*

Modelling the train with timing assumption

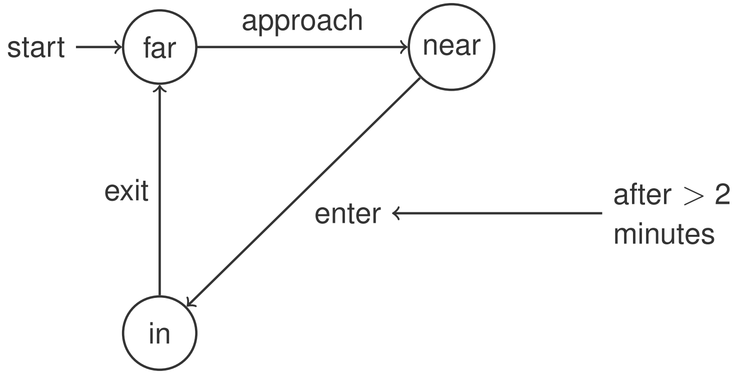


Figure: Train model

Modelling the controller with timing assumptions

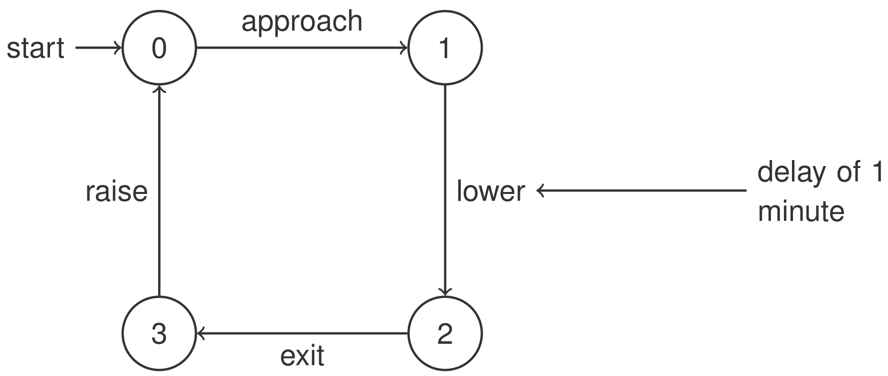


Figure: Controller model

Modelling the gate

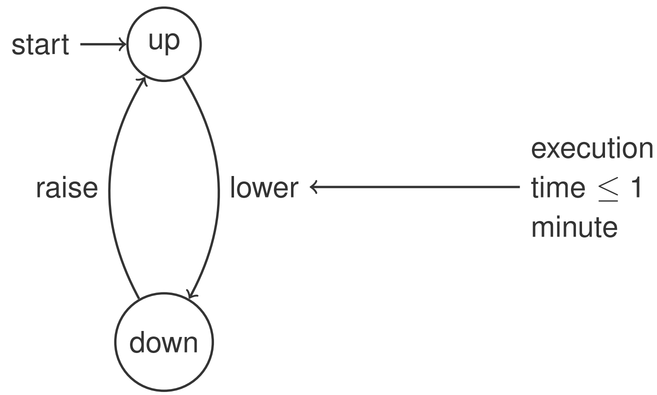


Figure: Gate model

Initial fragment of *Train* || *Controller* || *Gate* with timing assumptions

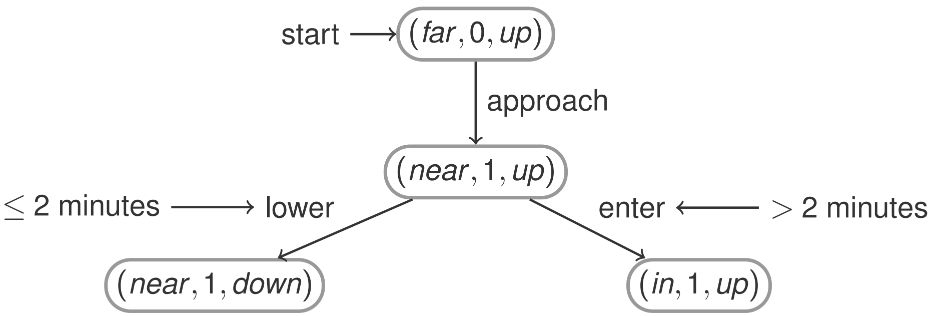


Figure: Initial fragment of *Train* || *Controller* || *Gate* with timing assumptions

Timed automatons

Timed automatons $=$ Finite automatons $+$ Clock constraints $+$ Clock resets

Clock constraints

Definition

A clock constraint over a set C of clocks is formed according to the grammar

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g \mid true$$

where $c \in \mathbb{N}$ is a constant and $x \in C$ is a clock variable. Let $CC(C)$ denote the set of clock constraints over C.
Clock constraints that do not contain any conjunctions are called *atomic*. Let $ACC(C)$ denote the set of all atomic clock constraints over C.

Remark

This definition ensures that all clock constraints form *convex sets,* a restriction that is often not necessary, but convenient.
We can, at the cost of a more involved theory, also admit *clock difference constraints* $x - y < c$ (this is allowed in Uppaal).

Timed automaton

Definition

A *timed automaton* is a tuple $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$, where

- Loc is a finite set of locations,
- $Loc_0 \subset Loc$ is a set of initial locations,
- Act is a finite set of actions,
- C is a finite set of clocks,
- $\hookrightarrow \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ is a transition relation,
- AP is a finite set of atomic propositions, and
- $L : Loc \to 2^{AP}$ is a labelling function

Handshaking for timed automatons

Let $TA_i = (Loc_i, Act_i, C_i, \hookrightarrow_i, Loc_{0,i}, Inv_i, AP_i, L_i)$ for $i \in \{1,2\}$ be two timed automatons, $H \subseteq Act_1 \cap Act_2$ and with $C_1 \cap C_2 = \emptyset$ and $AP_1 \cap AP_2 = \emptyset$. The timed automaton $TA_1 \parallel_H TA_2$ is defined by:

$$TA_1 \parallel_H TA_2 = (Loc_1 \times Loc_2, Act_1 \cup Act_2, C_1 \cup C_2, \hookrightarrow, \\ Loc_{0,1} \times Loc_{0,2}, Inv, AP_1 \cup AP_2, L)$$

where

- $L((\ell_1, \ell_2)) = L_1(\ell_1) \cup L_2(\ell_2)$
- $Inv((\ell_1, \ell_2)) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$

- For $\alpha \in H$:

$$\frac{\ell_1 \xhookrightarrow{g_1:\alpha,D_1}_1 \ell_1' \qquad \ell_2 \xhookrightarrow{g_2:\alpha,D_2}_2 \ell_2'}{(\ell_1, \ell_2) \xhookrightarrow{g_1 \wedge g_2:\alpha,D_1 \cap D_2} (\ell_1', \ell_2')}$$

- For $\alpha \notin H$:

$$\frac{\ell_1 \xhookrightarrow{g_1:\alpha,D_1}_1 \ell_1'}{(\ell_1, \ell_2) \xhookrightarrow{g_1:\alpha,D_1} (\ell_1', \ell_2)}$$

and

$$\frac{\ell_2 \xhookrightarrow{g_1:\alpha,D_2}_2 \ell_2'}{(\ell_1, \ell_2) \xhookrightarrow{g_2:\alpha,D_2} (\ell_1, \ell_2')}$$

Clock evaluations

Definition

A *clock evaluation* η for a set C of clocks is a function $\eta : C \to \mathbb{R}_{\geq 0}$, assigning each clock x each current value $\eta(x)$.
Let $Eval(C)$ denote the set of all clock evaluations for C.

Remark

For dense time model, a clock valuation is actually a function $\eta : C \to \mathbb{Q}_{\geq 0}$.

Satisfaction relation for clock constraints

Definition

For set C of clocks, $x \in C$, $\eta \in Eval(C)$, $c \in \mathbb{N}$, and $g, g' \in CC(C)$, let $\models \subseteq Eval(C) \times CC(C)$ be defined by:

- $\eta \models true$
- $\eta \models x < c$ if and only if $\eta(x) < c$
- $\eta \models x \leq c$ if and only if $\eta(x) \leq c$
- $\eta \models x > c$ if and only if $\eta(x) > c$
- $\eta \models x \geq c$ if and only if $\eta(x) \geq c$
- $\eta \models g \wedge g'$ if and only if $\eta \models g$ and $\eta \models g'$

Clock translation and reset

Definition

A *clock translation* is a function $(\eta + d)$, where $d \in \mathbb{N}$, such that for all $x \in C$: $(\eta + d)(x) = \eta(x) + d$.

Definition

A *reset* on a clock x, written reset x in η, is a clock valuation given by:

$$(\text{reset } x \text{ in } \eta)(y) \begin{cases} \eta(y) & \text{if } y \neq x \\ 0 & \text{if } y = ex \end{cases}$$

Nested resets can be abbreviated: (reset x in (reset y in η)) can be written as reset x, y in η.

Observe that translations and resets are associative and commutative, e.g. $(\eta + d) + e = (\eta + e) + d$.

Transition system semantics of a timed automaton

Let $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ be a timed automaton. Its transition system $TS(TA) = (S, Act', \to, I, AP', L')$ is given by:

- $S = Loc \times Eval(C)$
- $Act' = Act \cup \mathbb{R}_{\geq 0}$
- $I = \{(\ell_0, \eta) \mid \ell_0 \in Loc_0 \wedge \forall x \in C : \eta(x) = 0\}$
- $AP' = AP \cup ACC(C)$
- $L'((\ell, \eta)) = L(\ell) \cup \{g \in ACC(C) \mid \eta \models g\}$

The transition relation is defined by two rules:

- *discrete transition:* $(\ell,\eta) \xrightarrow{\alpha} (\ell',\eta')$ if the following conditions hold:
  1. There is a transition $\ell \xhookrightarrow{g:\alpha,D} \ell'$ in TA
  2. $\eta \models g$
  3. $\eta' =$ reset D *in* η
  4. $\eta' \models Inv(\ell')$
- *delay transition:* $(\ell,\eta) \xrightarrow{d} (\ell,\eta+d)$ for $d \in \mathbb{R}_{>0}$ provided that $\eta + d \models Inv(\ell)$.

Guards vs. invariants

- A *guard* expresses a condition under which a transition can be taken. They limit progress.
- An *invariant* expresses a condition under which it is allowed to delay in the current state. They force progress.

Time convergence

Consider a location ℓ such that for any $t < d$ (where d is a constant $d \in \mathbb{R}_{>0}$), the clock valuation $\eta + t \models Inv(\ell)$. A possible execution fragment starting from this location is given by

$$(\ell,\eta) \xrightarrow{d_1} (ell,\eta+d_1) \xrightarrow{d_2} (ell,\eta+d_1+d_2) \xrightarrow{d_3} (ell,\eta+d_1+d_2+3)\ldots$$

where $\lim_{j\to\infty} \sum_{i=1}^{j} d_i = d$. Such an infinite path fragment is called *time convergent*.
Time convergent paths are not realistic and should be ignored.

Preparatory exercise

- Download and install Uppaal from `http://www.uppaal.com/`
- Try to model today's examples and show some basic properties
- Enjoy the season

Suggested reading

- (Baier & Katoen, 2008), Chapter 9

Timed Computation Tree Logic

Timed Computation Tree Logic

- Starting from the definition of a transition system of a timed automaton, we obtain execution fragments and executions in the same manner as for transitions systems.
- This time, we obtain *timed traces*, i.e. sequences of pairs (s, η), where s is a state and η is a clock valuation.
- Timed Computation Tree Logic (TCTL) is a real-time variant of CTL aimed to express properties of timed automatons.

Timed Computation Tree Logic

Definition

Formulae of TCTL are generated by the following grammar for *state formulae*:

$$\Phi ::= true \mid a \mid g \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists\phi \mid \forall\phi \quad ,$$

where $a \in AP$ and $g \in ACC(C)$ and φ is a *path formula* given by:

$$\phi ::= \Phi \mathscr{U}^J \Phi$$

where $J \subset \mathbb{R}_{\geq 0}$ is an interval whose bounds are natural numbers.

We define $\Diamond^J \Phi \triangleq true \mathscr{U}^J \Phi$ and

$$\exists\Box^J \Phi \triangleq \neg\forall\Diamond^J \neg\Phi \text{ and } \forall\Box^J \Phi \triangleq \neg\exists\Diamond^J \Phi$$

Timed Computation Tree Logic

Abbreviations

Write $\leq c$ for $[0,c]$, $< c$ for $[0,c)$, $> c$ for (c,∞), $\geq c$ for $[c,\infty)$, and nothing for $[0,\infty) = \mathbb{R}_{\geq 0}$

Example

The property "the light cannot be continuously switched on for more than 2 minutes" is expressed by the TCTL formula

$$\forall\Box(on \rightarrow \forall\Diamond^{\geq 2}\neg on$$

Sets of path fragments

Let *TA* be a timed automaton. Let $\pi = s_0 \overset{d_0}{\Rightarrow} s_1 \overset{d_1}{\Rightarrow} s_2 \overset{d_2}{\Rightarrow} s_3 \overset{d_3}{\Rightarrow} \dots$ be the equivalence class of all infinite execution fragment in $TS(TA)$ of the form $(\ell_0,\eta_0) \xrightarrow{d_0^1} \dots \xrightarrow{d_0^{k_0}} (\ell_0,\eta_0+d_0) \xrightarrow{\alpha_0} (\ell_1,\eta_1) \xrightarrow{d_1^1} \dots \xrightarrow{d_1^{k_1}} (\ell_1,\eta_1+d_1) \xrightarrow{\alpha_1} (\ell_2,\eta_2)\dots$.
That means: The set of path fragments represents the passage of d_0 time units by all finite sequences of delay transitions which let d_0 time units pass. *Time divergent* execution fragments that perform only a finite number of actions are represented by a set of path fragments that ends in $(\ell_m,\eta_m) \overset{1}{\Rightarrow} (\ell_m,\eta_m+1) \overset{1}{\Rightarrow} (\ell_m,\eta_m+2) \overset{1}{\Rightarrow} \dots$

Semantics of TCTL

Let *TA* be a timed automaton, let state $\ell \in Loc$, $a \in AP$, $g \in ACC(C)$, $J \subseteq \mathbb{R}_{\geq 0}$ an interval, φ,ψ TCTL state formulae, and ξ be a path formula. Let $Paths(s)$ be the set of all infinite execution fragments of T that start in s.

- $(\ell,\eta) \models a$ if and only if $a \in \mu(\ell)$
- $(\ell,\eta) \models g$ if and only if $\eta \models g$
- $(\ell,\eta) \models \neg\varphi$ if and only if $(\ell,\eta) \not\models \varphi$
- $(\ell,\eta) \models \varphi \wedge \psi$ if and only if $(\ell,\eta) \models \varphi$ and $(\ell,\eta) \models \psi$
- $(\ell,\eta) \models \exists\xi$ if and only if for some $\pi \in Paths_{div}((\ell,\eta))$, $\pi \models \xi$.
- $(\ell,\eta) \models \forall\xi$ if and only if for some $\pi \in Paths_{div}((\ell,\eta))$, $\pi \models \xi$.
- $s \models \forall\xi$ if and only if for every $\pi \in Paths(s)$, $\pi \models \xi$.

Semantics for path formulae

For a time-divergent path $\pi \in (\ell_0,\eta_0) \overset{d_0}{\Rightarrow} (\ell_0,\eta_0+d_0) \overset{d_1}{\Rightarrow} (\ell_1,\eta_1+d_1) \overset{d_2}{\Rightarrow} \dots$, the satisfaction relation is defined by: $\pi \models \Phi\mathcal{U}^J\Psi$ if and only if

- there exists $i \geq 0$ such that $s_i + d \models \Psi$ for some $d \in [0,d_i]$ with $d + \sum_{k=0}^{i-1} d_k \in J$ and
- for all $j \leq i$ we have $s_j + d' \models \Phi \vee \Psi$ for any $d' \in [0,d_j]$ with $d' + \sum_{k=0}^{j-1} d_k \leq d + \sum_{k=0}^{i-1} d_k$

Semantics on transition systems

Definition

Given a timed automaton TA, the satisfaction set $SAT_{TA}(\varphi)$ for a TCTL formula φ is defined by:

$$SAT_{TA}(\varphi) \triangleq \{(\ell,\eta) \in Loc \times Eval(C) \mid (\ell,\eta) \models \varphi\}.$$

The timed automaton TA satisfies a TCTL formula φ if and only if φ holds in all initial states of $TS(TA)$:

$$TA \models \varphi \text{ if and only if } \forall \ell_0 \in I : (\ell_0, \eta_0) \models \varphi$$

where $\eta_0(x) = 0$ for all $x \in C$.

TCTL vs. CTL

Any TCTL formula Φ in which all intervals are of the form $[0,\infty)$ may be considered to be a CTL formula. But we can find an example in which

$$TA \models_{TCTL} \Phi \text{ and } TS(TA) \not\models_{CTL} \Phi \quad !$$

The reason is: the semantics of TCTL is restricted to only time divergent paths, whereas we also consider time convergent paths for CTL.

Lifeness and time-lock

Lemma

A timed automaton is time-lock-free if and only if for all $(\ell,\eta) \in Reach(TS(TA))$ $s \models \exists\Box true$.

Note: this is different from $TA \models \forall\Box\exists\Box true$!

TCTL model checking

Problem

We cannot use CTL model checking algorithms:

$$TA \models \Phi \text{ if and only if } TS(TA) \models \Phi$$

and $TS(TA)$ is an *infinite* transition system.

Solution

$$TA \models \Phi \text{ if and only if } RTS(TA,\Phi) \models \hat{\Phi}$$

where $RTS(TA,\Phi)$ is a finite transition system called *region automaton* and $\hat{\Phi}$ is a CTL formula obtained from Φ as explained next.

Eliminating timing parameters

First we express all $J \neq [0,\infty)$ in the TCTL formula by atomic clock constraints.
Idea: For each $J \neq [0,\infty)$ introduce a *fresh* clock z that neither occurs in Φ nor in *TA* and add a clock constraint to Φ that uses z.

Elimination theorem

Define the timed automaton $TA \oplus z$ by adding z to the clocks. Then:

1. $(\ell,\eta) \models_{TCTL} \exists(\Phi \mathcal{U}^J \Psi)$ if and only if $(\ell,\eta \cup \{z \mapsto 0\}) \models_{TCTL} \exists((\Phi \vee \Psi)\mathcal{U}(\Psi \wedge z \in J)$
2. $(\ell,\eta) \models_{TCTL} \forall(\Phi \mathcal{U}^J \Psi)$ if and only if $(\ell,\eta \cup \{z \mapsto 0\}) \models_{TCTL} \forall((\Phi \vee \Psi)\mathcal{U}(\Psi \wedge z \in J)$

Region automatons

Idea

We cannot use $TS(TA)$, because this is an infinite transition system. We'd like to find an appropriate equivalence relation on clocks, written $\simeq$, such that:

- Equivalent clock valuations should satisfy the same clock constraints:

$$\eta \simeq \eta' \implies (\eta \models g \text{ iff } \eta' \models g \text{ for all } g \in ACC(TA) \cup ACC(\Phi))$$

- Time-divergent paths emanating from equivalent states should be "equivalent". This guarantees that equivalent states satisfy the same path formulae.
- The number of equivalent classes under $\simeq$ is finite.

Notations

- Integral part of a real number: $\lfloor 3.14 \rfloor = 3$
- Fractional part of a real number: $frac(3.14) = 0.14$

Observation

Clock valuations "look" equivalent if we define $\simeq$ by:

$$\forall x : \lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor \text{ and } frac(\eta(x)) = 0 \mathit{iff} frac(\eta'(x)) = 0$$

This relation is too coarse!

Clock equivalence

Definition

Let *TA* be a timed automaton, Φ a *TCTL* formula both over the same set C of clocks, and c_x the largest constants with which $x \in C$ is compered to in Φ.
Clock valuations are *clock-equivalent* if and only if either:

- for any $x \in C$ it holds that $\eta(x) > c_x$ and $\eta'(x) > c_x$
- for any $x, y \in C$ with $\max\{\eta(x), \eta'(x)\} \leq c_x$ and $\max\{\eta(y), \eta'(y)\} \leq c_y$ all the following conditions hold:
  - $\forall x : \lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$ and $frac(\eta(x)) = 0$ *iff* $frac(\eta'(x)) = 0$
  - $frac(\eta(x)) \leq frac(\eta(y))$ if and only if $frac(\eta'(x)) \leq frac(\eta'(y))$

Number of regions

Theorem

The number of clock regions is bounded by:

$$|C|! \cdot \prod_{x \in C} \max\{c_x, 1\} \leq |Eval(C)/\simeq| \leq |C|! \cdot 2^{|C|-1} \cdot \prod_{x \in C} (2\max\{c_x, 1\} + 2)$$

Decidable and undecidable problems of timed automatons

Complexity and decidability results

- Timed automatons test the border of model checking techniques: they are a form of infinite state systems that have a finite representation.
- A finite representation is not enough for decidable procedures: a common example is a Turing machine

Decidability results

Proof in (Alur & Dill, 1994, Theorem 4.17):

Theorem

The emptiness problem for timed automatons in which the coefficients of the guards are rational numbers is PSPACE-complete.

And Puri shows (Puri, 1999):

Theorem

The reachability problem for timed automatons in which the coefficients of the guards are from the set $\{1, \sqrt{2}\}$ is undecidable.

More results: see (Henzinger, Kopke, Puri, & Varaiya, 1998).

Proof idea I

Definition

A *counter machine* $C = (V, L, E, \ell_0)$ has a

- a set of counters V
- set of locations L,
- a set of transitions $E \subseteq L \times Act(V) \times L$, where $Act(V)$ is given by the grammar:

$$A ::= inc(v) \mid dec(v) \mid reset(v) \mid (v = 0) \mid (v \neq 0)$$

- An initial location ℓ_0
- Terminal control location do not have any outgoing edges.

States are from $L \times (V \to \mathbb{N}_0)$ and runs are defined in the obvious way.

Proof idea II

Theorem

A 2 counter machine (one with $|V| = 2$) can simulate a Turing machine.

Corollary

It is not decidable whether a 2 counter machine reaches a terminal location.

Proof idea III

We formulate the reachability problem for 2 counter machines to the reachability problem of a timed automaton.

Theorem

For any 2 counter machine M we find a timed automaton TA with all coefficients taken from $\{1, \sqrt{2}\}$ such that M reaches a terminal location if and only if TA reaches a designated "final state".

Encoding a counter

The value k of a counter will be stored as a real number in the interval $[0,1)$ using the following encoding:

$$e(k) \triangleq k\sqrt{2} - \lfloor k\sqrt{2} \rfloor$$

Lemma

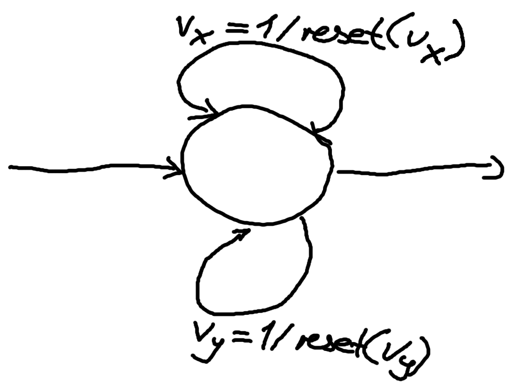
The encoding e is injective.

In the timed automaton, two clocks v_x, v_y are used to store and manipulate the value of the counter v

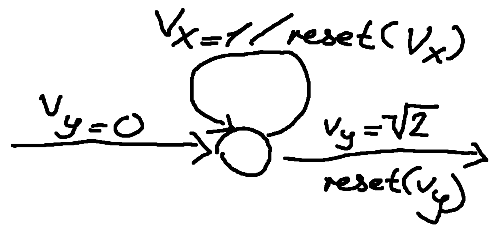
Encoding the transitions

- We introduce three gadgets that encode the actions:
  - A maintain gadget with: v_x has the value $e(v)$ if $v_y = 0$
  - An increment gadget with: v_x has the value $e(v+1)$ if $v_y = 0$
  - A decrement gadget with: v_x has the value $e(v-1)$ if $v_y = 0$
- The guards can be expressed by:
  - $(v = 0)$ becomes $v_x = 0 \wedge v_y = 0$
  - $(v \neq 0)$ becomes $0 < v_x \wedge v_x < 1 \wedge v_y = 0$

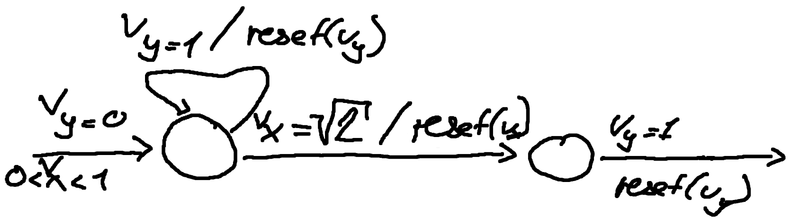
Maintain gadget



Increment gadget



Decrement gadget



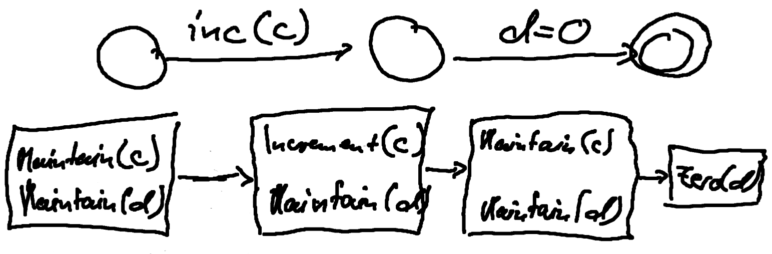
From 2 counter machine to transition

– The timed automaton is formed by composing a gadget for each counter according to the description of the counter machine, e.g.: $inc(c)$ is formed by composing

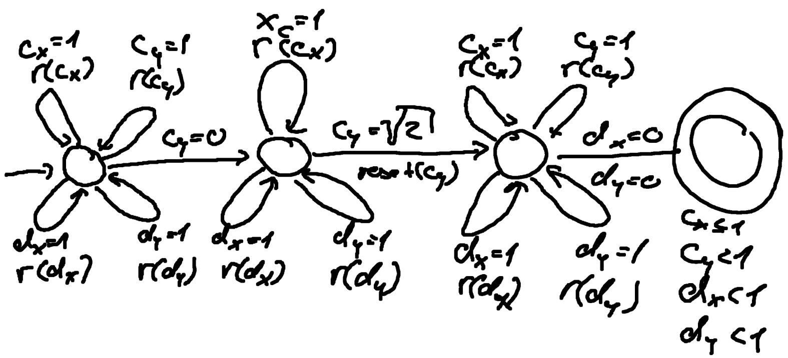
$$Maintain(c) \mid Maintain(d); Increment(c) \mid Maintain(d)$$

Each operation is preceded by a maintain phase to make sure that no time passed before we execute the operation.

Example



Example as timed automaton



Decidability and clock rates

Theorem

The emptiness problem is undecidable for 2-rate timed automata and rational coefficients.

Proof.

We can encode a counter machine as a timed automaton with 3 clocks at different rates. Suppose we have three clocks, one with rate 1 and two with rate 2. Then we can encode the values of two counters in the i-th machine configuration by the values of clocks x_1 and x_2 at accurate time i: the counter value k is encoded by the clock value $1/2^k$. The increment and decrement can be formed as before, but uses only the guards 1. □

Model checking software

Model checking software

Software is usually a really detailed model of a process.

- Often, more than one task is performed by a program.
- It is not well-structured
- Works with lots of data.

Challenge

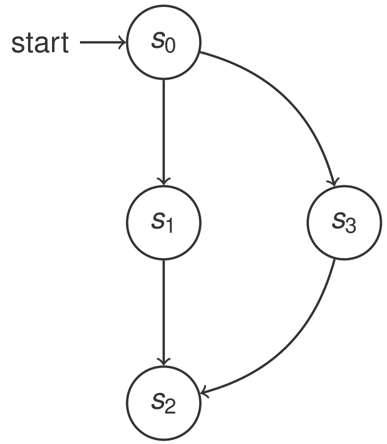
- Data increases state space, especially when stored.
- Control flow contains activities not relevant to the property
- Spurious distinctions

The Intuition of Model Checking

- Compute whether a system satisfies a certain behavioural property:
  - Is the system deadlock free?
  - When ever a packet is sent will it eventually be received?
- Unlike *testing,* all possible behaviours of a system are analysed
- Model checking is automatic if the system is finite state
  - Potential for being a *push-button* technology
  - Almost no expert knowledge required

System description

All algorithms operate on *transition systems* or *region transition systems*



Each transition represents an execution step

Each state represents all variable values and location counters

The labels represent predicates in each state, e.g. $x \leq 5$

Specifying properties

Temporal logic

- Express properties of event orderings in time
- “Always” when a packet is sent it will “eventually” be received

Linear time

- Every moment has a unique successor
- Infinite sequences (words)
- Linear-time temporal logic (LTL)

Branching time

- Every moment may have several successors
- Infinite trees
- Computation tree logic (CTL)

Safety and liveness

Safety properties
- Invariants, deadlocks, reachability, etc.
- Can be checked on finite traces
- “something bad never happens”

Liveness properties
- Fairness, response, etc.
- Can only be checked on infinite traces
- “something good will eventually happen”

Model checking

- Given a transition system $M = (S, I, \rightarrow, L)$ that represents a finite-state concurrent system and a temporal logic formula φ expressing some desired specification, fine the set of states that satisfy φ: $\{s \in S \mid M, s \models \varphi\}$
- Normally, some states of the concurrent system are designated as initial states. The system satisfies the specification provided all initial states are in the set. We then write $M \models \varphi$

Explicit state vs. symbolic model checking

Explicit state

- States are enumerated on-the-fly
- Forward-analysis
- Stores visited states in a hashtable

Characteristics

- Memory intensive
- Good for finding concurrency errors
- Can deal well with long executions
- Can handle dynamic creation of threads/objects
- Mostly used for software

Symbolic

- Sets of states are manipulated at a time
- Typically a backwards analysis
- Transition relation encoded by BDDs

Characteristics

- Can handle very large state spaces
- Not as good for asynchronous systems
- Cannot deal well with long executions
- Works best without dynamic thread/object creation
- Mostly used for hardware

Program model checking

Why analyse programs with model checkers?

- Designs and models of programs are hard to come by, but buggy programs are everywhere
- Testing is inadequate for complex programs (threads, pointers, objects, etc.)
- Static program analysis was already an established field, mostly in compiler optimisation and now also in verification

Problem

Most model checkers cannot deal with the features of modern programming languages

- Adapting programs to model checkers using translation and abstraction
- Bringing model checking to programs by improving algorithms to deal with them directly

Abstraction is the enabling technology

Program

```
void add(Object& o) {
  buffer[head++]=o;
  head %= size;
}
```

Model checker input

The need for abstraction

- Model checkers don't take real "programs" as input
- Model checkers typically work on finite state systems
- Abstraction therefore solves two problems:
  - It allows model checkers to analyse a notation they could not before
  - Reduces the size of the state space to something manageable
- Abstraction comes in three flavours

Over-approximation more behaviour is displayed by the abstracted system than is present in the original

Under-approximation less behaviour is displayed by the abstracted system than is present in the original

Precise abstraction the same behaviour is displayed by both the abstracted and the original program

Remark

The term "behaviour" depends on what we observe!

Under-approximation

- Remove parts of the program deemed "irrelevant" to the property being checked
  - Restrict input values to $[0, 10]$ rather than all integer values
  - Queue size of 3 instead of unbounded
- The abstraction of choice in the early days of program model checking
  - Used during the translation of code to a model checkers input language
  - Typically manual
  - No guarantee that the right behaviours are removed

Lemma

Let φ be a universal property (LTL formula or $\forall$-CTL formula). If $TS_1 \leq_L TS_2$ and $TS_1 \not\models \varphi$, then $TS_2 \not\models \varphi$.

Observation

Under-approximation is good for systematic testing and finding bugs, but not for verifying correctness.

Cone of influence reduction

- A *precise* abstraction with respect to the property being checked
- May be computed automatically:
  1. Given a property φ, let V be the set of variables used in φ
  2. Mark all control-flow statements (if, while, goto) and add all variables read in these statements to V
  3. Mark all statements that *write* to variables in V
  4. Add all variables read in the marked statements to V
  5. Repeat at 3 until no new statement is marked
  6. Delete all unmarked statements

Over-approximations

- Maps sets of states in the original program to one state in the abstracted program
  - Reduces the number of states, but increases the number of possible transitions, and hence the number of behaviours
  - Can in rare cases lead to a precise abstraction
- Type-based abstractions, e.g. replace int by sign abstraction $\{neg, pos, zero\}$
- Predicate abstraction: Replace predicated in the program by Boolean variables, and replace each statement that modifies the validity of the predicate with a corresponding statement that modifies the Boolean.
- Automated (conservative) abstraction (bit-state hashing)
- Eliminating spurious errors is the big problem:
  - Abstract program has more behaviour, therefore when an error is found in the abstract program, is it also an error in the original program?
  - Most research focuses on the problem of eliminating spurious errors and finding counter examples in the original program. Keyword: *abstraction refinement*

Bringing model checking to programs

- Allow model checkers to take modern programming languages as input
- Major hurdle is how to encode the state of the system efficiently
- Alternatively state-less model checking (no state encoding and storing)
- Almost exclusively explicit-state model checking
- Abstraction can still be used as well (source-to-source abstractions, slicing, . . .)

Hand translation

- Translate a program into the model checker's input language by hand
- Apply abstractions as necessary
- Labour-intensive and error-prone
- Confidence by theorem proving, but often proving the abstraction is similar to proving the property

Examples

- Remote agent (Lowry, Havelund, & Penix, 1997)
  - Translation from LISP to Promela
  - Heavy abstraction (essentially a remodelling)
  - 3 man months
- DEOS (1998/1999), see (Penix et al., 2005)
  - C++ to Promela (most effort in environment generation)
  - Limited abstraction (programmers produced sliced systems)
  - 3 man months

Semi-automatic translation

- Table-driven translation and abstraction
- User specifies code fragments in C and how to translate them to Promela (note: SPIN now accepts Promela with fragments of C)
- Translation is then automatic
- Advantages
  - Can be reused when program changes
  - Works well for programs with long development and only local changes

Fully automatic translation

- Advantage: No human intervention required
- Disadvantage: limited by capabilities of target system
- Examples:
  - Java PathFinder 1 (translates from Java to Promela)
  - JCAT (translates from Java to Promela)
  - Bandera (today: Bogor) (translates from Java to Promela, SMV)

Remark

An extension of SPIN that supports Process deletion is available and called dSPIN.

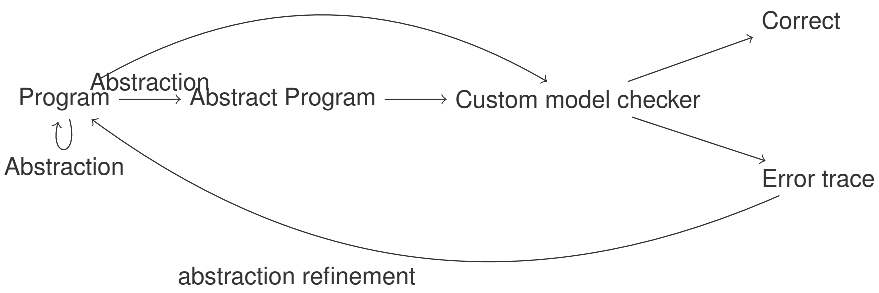
Partial-order reductions

- Reduce the number of interleavings of independent concurrent transitions

Basic ideas

- Independence: Independent transitions cannot disable nor enable each other. Enabled independent transitions are commutative.
- Partial-order reductions mostly apply during the on-the-fly construction of transition systems
- Based on a selective search principle: compute a subset of enabled transitions in a state to execute
  - ample sets (reduced transitions)
  - persistent sets (reduced states)
- See Chapter 8 of the book (Baier & Katoen, 2008)

Structure of a software model checker



- Custom-made model checkers for programming languages with automatic abstraction at the source code level
- Automatic abstraction and translation based transformation to new "abstract" formalism for model checker
- Abstraction refinement mostly automated

Data abstraction

Problem

Programs use data types with possibly infinite domain (e.g., lists). Use data abstraction and abstract interpretation (ACM, 1977) to build a finite model.

Solutions

- Data type based abstractions:
  - For each data type define an *abstract domain*, e.g. sign abstraction for integers $\{neg, zero, pos\}$.
  - Replace all operations with corresponding abstract operations:
    - $add(pos, pos) = pos$
    - $sub(pos, pos) = pos \mid zero \mid neg$
    - $eq(pos, pos) = true \mid false$
- Predicate abstraction (Graf & Saïdi, 1997): Create abstract state space from the predicates defined in the concrete system

Abstract interpretation and data type abstraction

Features of abstract interpretation

- Each data type has a *concrete* domain, e.g. **`int`**'s is the integer numbers. The operations have their natural meaning.
- A mapping from the concrete domain to an *abstract domain*. Each operation is interpreted by an abstract operation on the abstract domain.
- Interpretation computes an over-approximation. Termination is guaranteed by a meet operator $\sqcup$ and an *extrapolation operator* ∇.

$$\alpha(x) = \begin{cases} \text{neg} & \text{if } x < 0 \\ \text{zero} & \text{if } x = 0 \\ \text{pos} & \text{if } x > 0 \end{cases}$$

$\odot$	neg	zero	pos
neg	pos	zero	neg
zero	zero	zero	zero
pos	neg	zero	pos

Abstract interpretation and model checking

- Abstract interpretation is a method to compute properties of a program based on data abstractions
- Additional constraints are placed on the relation between concrete and abstract domain:
  - Both must be partially ordered with least element $\perp$ (also called lattice) (e.g., concrete integers are enriched by $\perp$ with $z \geq \perp$ for all $z \in Z$)
  - Abstract and concrete domain must form a *Galois connection*: There is a pair of functions $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$ s.t. $c\alpha \leq a$ iff $c \leq a\gamma$
  - The partial order on the domains induces two additional operations $\sqcup$ (supremum, join) and $\sqcap$ (infimum, meet)
  - Loops are abstracted with help of the acceleration operator $a \nabla b = (a \sqcup b) \sqcup b \cdots$
- Abstract interpretation computes a safe over-approximation
- Problem: choice of abstract domain. Shall we use intervals or signs to represent integers?

Infeasible counter examples

When over-approximating, we may have infeasible counter examples

- Abstraction refinement is used to refine the abstraction when an infeasible counter example is found
- This presupposes an ordered set of data abstractions
- When a counter example is infeasible, compute why it is infeasible and use the information to choose the next refined data abstraction
- Try again with the new abstraction

Soundness of abstraction

- Infeasible counter examples are the result of two problems:
  1. The data abstraction introduces more transitions, i.e. more non-determinism and new computations not possible in the original program
  2. The data abstraction does not preserve the property:

     $\Box(x \leq 5)$ is abstracted to $\Box(x = pos \vee x = zero \vee x = neg)$

     where the abstracted property is a tautology.
     - Must prove that the abstraction is *sound*, i.e.:

       $M\alpha \models \varphi\alpha$ implies $M \models \varphi$
- This topic has been worked out well by Dennis Dams in (Dams, 1996).

Predicate abstraction (Graf & Saïdi, 1997)

Idea

Create abstract state space from the predicates defined in the concrete system

- Identify all relevant predicates in the program.
- Replace each predicate by a Boolean variable.
- Replace each statement by a corresponding assignment to the boolean variables.

Example

```
int x = 0;
int y = 0;
while (x==y) {
  x++;
  if (x>0) {
    ++y;
  }
}
```

```
bool a = true;
bool b = false;
while (a) {
  a = false; b = true;
  if (b) {
    a=true;
  }
}
```

When verification fails

Use abstraction refinement to add more information. Difficult, because we need to guess additional Boolean variables that provide suitable distinctions.

Divide, abstract, and model-check

1. Huge systems must be suitable decomposed into different parts.
2. Since parts do not run individually, we need to have a *model* of the environment (corresponds to a *precondition* in Hoare logic)
3. Each part may still be too large for model checking, so the part needs to be abstracted. (Abstraction function and composition operators must be monotonic: $C \sqsubseteq A \implies C \circ P \sqsubseteq A \circ P$ and abstraction function must be *sound!*)
4. Model check properties of each part
5. Use model checking or theorem proving to combine the results.

Software model checkers for Java

- Java Pathfinder (`http://babelfish.arc.nasa.gov/trac/jpf`)
- Bogor (`http://bogor.projects.cis.ksu.edu/content/view/88/54/`)

Software model checkers for C

- Verisoft (`http://cm.bell-labs.com/who/god/verisoft/`)
- CBMC (`http://www.cprover.org/cbmc/`) (only bounded model checking)
- BLAST (`http://mtc.epfl.ch/software-tools/blast/index-epfl.php`)
- SLAM (`http://research.microsoft.com/en-us/projects/slam/`) used as part of driver certification

Conclusion:

The road ahead

What did we look at?

- The model checking problem ($M \overset{?}{\models} \varphi$)
- Decidability guaranteed for finite state models M and decidable properties φ
- Models are represented in textual languages (SPIN, SMV) and graphical languages (UPPAAL)
- Models are interpreted by transition systems (including region transition systems) or binary decision diagrams (BDD)
- Properties are specified in subsets of CTL^* (LTL, CTL, $\forall$-CTL, TCTL)

Decision procedures

- Looked at basic model checking algorithms
  - Büchi-automatons based for LTL
  - Symbolic for CTL

State-space reduction methods

- Partial-order reduction (automatic, sketched)
- Abstraction (simulation, bi-simulation, sketched)
- Data abstraction and abstract interpretation (mentioned)

Modelling and tools

Used all major, freely available model checkers

SPIN	on-the-fly, Büchi-automaton based, LTL
NuSMV	symbolic, BDD, CTL (and LTL)
UPPAAL	timed, TCTL

What did we not look at?

From the book, we did not look at

- Optimised versions of the model checking algorithms
- Correctness proofs for the model checking algorithms
- Computing (bi-)simulations automatically by partition refinement. See Section 6.x ff. of (Baier & Katoen, 2008).
- Partial-order reduction algorithms (ample-set, persistent-set, preserved properties). See Chapter 7 of (Baier & Katoen, 2008).

Going beyond this lecture

- State-space reduction techniques
  - Symmetry reduction (Ip & Dill, 1996)
  - Delta-encoding (d'Amorim, Lauterburg, & Marinov, 2007)
- Model-checking stochastic and probabilistic properties
  - Very different from conventional model checking, because here we look at quantitative reasoning (results are probabilities) and not qualitative reasoning (results are "yes" or "no and counter example")
  - See Chapter 9 of (Baier & Katoen, 2008).
- Hybrid systems (Alur et al., 1995)
  - Generalisation of timed systems, thus mostly undecidable problems
  - Important for understanding control systems
- The huge topic: Making it scale to real software systems

End

Bibliography I

Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P.-H., Nicollin, X., et al. (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, *138*(1), 3–34.

Alur, R., & Dill, D. L. (1994, April). A theory of timed automata. *Theoretical Computer Science*, *126*(2), 183–235.

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. Cambridge, MA: MIT Press.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computer*, *35*(8), 677–691.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., & Hwang, L. J. (1990). Symbolic model checking: $10^2$0 states and beyond. In *LICS* (pp. 428–439). IEEE Computer Society Press.

Bibliography II

Clarke, E. M., & Draghicescu, A. (1989). Expressibility results for linear time and branching time logics. In J. W. de Bakker, W.-P. de Roever, & G. Rozenberg (Eds.), (Vol. 354, pp. 428–437). Heidelberg: Springer-Verlag.

Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., et al. (1993). Verification of the Futurebus+ cache coherence protocol. In D. Agnew, L. J. Claesen, & R. Camposano (Eds.), *CHDL* (Vol. A-32, pp. 15–30). North-Holland.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. MIT Press.

Clarke, E. M., Long, D. E., & McMillan, K. L. (1989). Compositional model checking. In *LICS* (pp. 353–362). IEEE Computer Society Press.

Bibliography III

Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (pp. 238–252). New York, NY, USA: ACM Press.

d'Amorim, M., Lauterburg, S., & Marinov, D. (2007). Delta execution for efficient state-space exploration of object-oriented programs. In D. S. Rosenblum & S. G. Elbaum (Eds.), *ISSTA* (pp. 50–60). ACM Press.

Dams, D. (1996). *Abstract interpretation and partition refinement for model checking*. Unpublished doctoral dissertation, Technische Universiteit Eindhoven.

Graf, S., & Saïdi, H. (1997). Construction of abstract state graphs with PVS. In O. Grumberg (Ed.), *CAV* (Vol. 1254, pp. 72–83). Heidelberg: Springer-Verlag.

Bibliography IV

Henzinger, T. A., Kopke, P. W., Puri, A., & Varaiya, P. (1998, August). What's decidable about hybrid automata? *Journal of Computer and System Sciences*, *57*(1), 94–124.

Holzmann, G. J. (2003). *The SPIN model checker*. Boston, MA: Addison-Wesley.

Ip, C. N., & Dill, D. L. (1996, August). Better verification through symmetry. *Formal Methods in System Design: An International Journal*, *9*(1/2), 41–75.

ISO/IEC standard for information technology — microprocessor systems — Futurebus+ — logical protocol specification. (1994). (ISO/IEC 10857, IEEE 896.1-1994)

Kurshan, R. P., & McMillan, K. L. (1989). A structural induction theorem for processes. In *PODC* (pp. 239–247). New York, NY, USA: ACM Press.

Bibliography V

Kyas, M. (2001). Verifying a network invariant for all configurations of the Futurebus+ Cache Coherence Protocol. In R. Mayr (Ed.), *VEPAS* (Vol. 50, pp. 357–370). Amsterdam: Elsevier.

Lamport, L. (1976, January). Comments on 'a synchronisation anomaly'. *Information Processing Letters*, *4*(4), 88–89.

Lowry, M., Havelund, K., & Penix, J. (1997). Verification and validation of ai systems that control deep-space spacecraft. In Z. W. Ras & A. Skowron (Eds.), *ISMIS* (Vol. 1325, pp. 35–47). Heidelberg: Springer-Verlag.

Manna, Z., & Pnueli, A. (1995). *Temporal verification of reactive systems: Safety*. Springer-Verlag.

Bibliography VI

McMillan, K. L. (2000, May). A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, *37*(1–3), 279–309.

Olderog, E.-R., & Dierks, H. (2008). *Real-time systems: Formal specification and automatic verification*. Cambridge University Press.

Parikh, R. (2001). Language as social software. In J. Floyd & S. Shieh (Eds.), *Future pasts: The analytic tradition in twentieth century philosophy* (pp. 339–350). Oxford: Oxford University Press.

Penix, J., Visser, W., Park, S., Pasareanu, C., Engstrom, E., Larson, A., et al. (2005, March). Verifying time partitioning in the DEOS scheduling kernel. *Formal Methods in System Design*, *26*(2), 103–135.

Puri, A. (1999, May). An undecidable problem for timed automata. *Discrete Event Dynamic Systems*, *9*(2), 135–146.

Bibliography VII

Tassey, G. (2002, May). *The economic impacts of inadequate infrastructure for software testing* (Final Report No. 02-3). Gaithersburg, MD, USA: National Institute of Standards and Technology.

Wittgenstein, L. (1922). *Tractatus logico-philosophicus*. London: Kegan Paul, Trench, Trubner & Co.

Wolper, P., & Lovinfosse, V. (1989). Verifying properties of large sets of processes with network invariants. In J. Sifakis (Ed.), *CAV* (Vol. 407, pp. 68–80). Heidelberg: Springer-Verlag.