M2-CSA: Validation & Certification: Timed Automata, Abstract Interpretation

Jan-Georg Smaus

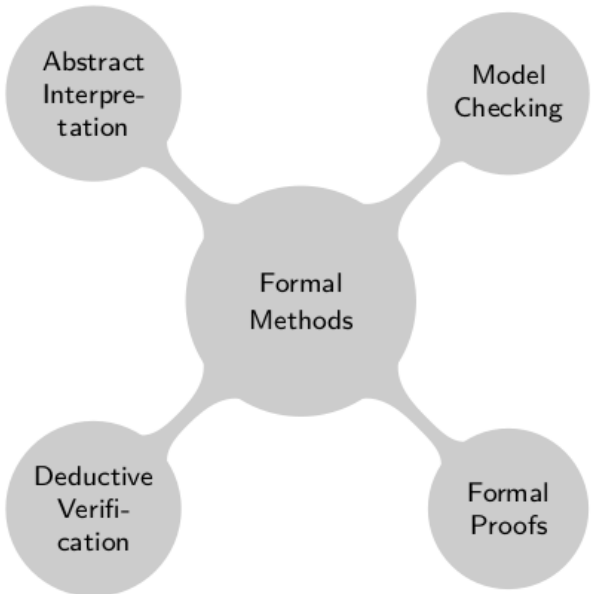
Université de Toulouse/IRIT

Year 2021/2022

Plan

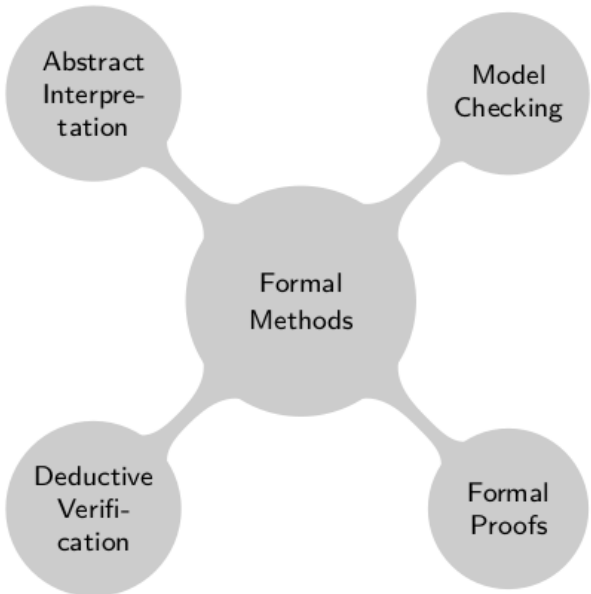
1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation

Introduction Lecture by Erik Martin-Dorel



"Jan-Georg Smaus: Model Checking / Deductive Verification"

Introduction Lecture by Erik Martin-Dorel



"Jan-Georg Smaus: Model Checking / Deductive Verification"
Plan: Abstract interpretation, timed automata modelling (not so much model checking; this was done during M1 in "Introduction to Embedded Systems").

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

Acknowledgements

The slides of this chapter are based on slides written by Mamoun Filali-Amine.

Real-time (timed) models

Recall:

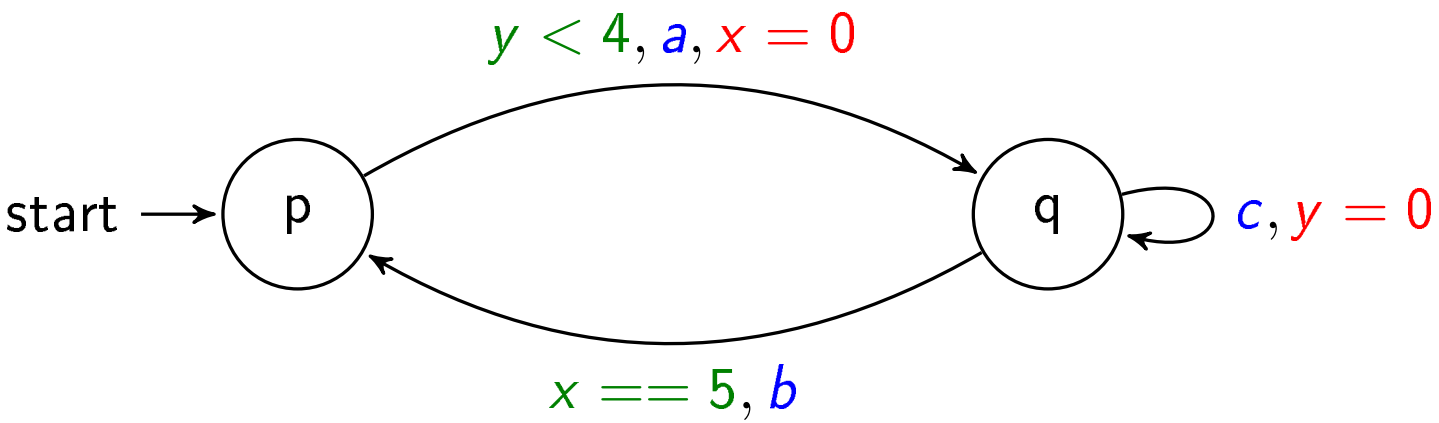
- Temporal models: we have a concept of time as a sequence of timepoints, i.e., the notion "X happens before Y".
- Timed or real-time models: we have a quantitative concept of time, i.e., events happen "at a certain time".

Here we consider real-time models exemplified by the formalism of timed automata presented in Uppaal style.

Exercise 1

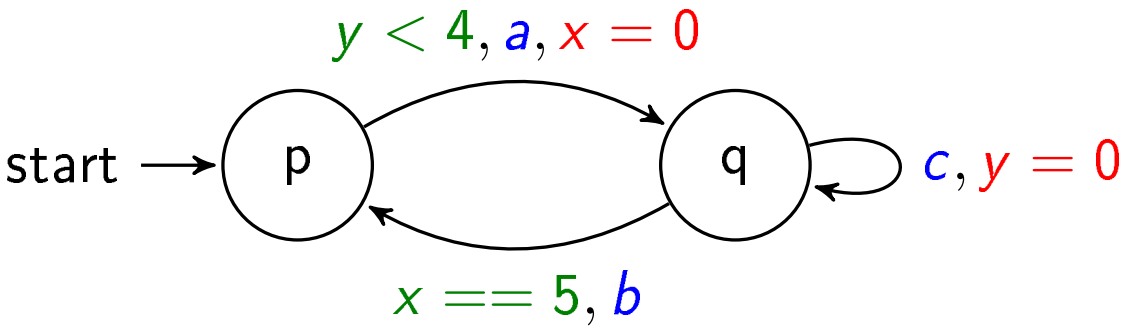
Where does the name "Uppaal" come from?

Example



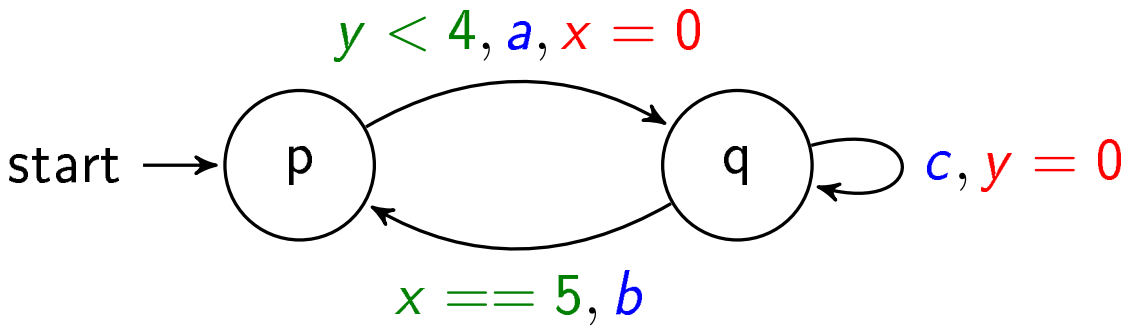
- locations (for usual automata: states): p, q.
- edges p $\rightarrow$ q, q $\rightarrow$ q, q $\rightarrow$ p.
  - edge p $\rightarrow$ q
    - guard:$y < 4$ *action* : a *reset* : $x = 0$
  - edge q $\rightarrow$ q
    - guard:**true** *action* : c *reset* : $y = 0$
  - edge q $\rightarrow$ p
    - guard:$x == 5$ *action* : b

Timed automaton run



A state is given by a location plus the values of all clocks, e.g. $\begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix}$.

Timed automaton run



A state is given by a location plus the values of all clocks, e.g. $\begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix}$.

A run is a sequence of alternating delay and discrete transitions:

$$\begin{pmatrix} p \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow[3.2]{delay} \begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix} \xrightarrow[a]{discrete} \begin{pmatrix} q \\ 0 \\ 3.2 \end{pmatrix} \xrightarrow[1.9]{delay} \begin{pmatrix} q \\ 1.9 \\ 5.1 \end{pmatrix} \xrightarrow[c]{discrete}$$

$$\begin{pmatrix} q \\ 1.9 \\ 0.0 \end{pmatrix} \xrightarrow[3.1]{delay} \begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix} \xrightarrow[b]{discrete} \begin{pmatrix} p \\ 5.0 \\ 3.1 \end{pmatrix}$$

Note that the guards are fulfilled at each discrete transition.

Timed automaton trace

Given a run, e.g.

$$\begin{pmatrix} p \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow[3.2]{\textit{delay}} \begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix} \xrightarrow[a]{\textit{discrete}} \begin{pmatrix} q \\ 0 \\ 3.2 \end{pmatrix} \xrightarrow[1.9]{\textit{delay}} \begin{pmatrix} q \\ 1.9 \\ 5.1 \end{pmatrix} \xrightarrow[c]{\textit{discrete}}$$

$$\begin{pmatrix} q \\ 1.9 \\ 0.0 \end{pmatrix} \xrightarrow[3.1]{\textit{delay}} \begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix} \xrightarrow[b]{\textit{discrete}} \begin{pmatrix} p \\ 5.0 \\ 3.1 \end{pmatrix}$$

a trace simply collects all the actions plus their absolute occurrence times. In this example: $(a, 3.2)(c, 5.1)(b, 8.2)$. We call such a sequence a timed word.

Each absolute occurrence time is simply the sum of all previous delays, e.g. $8.2 = 3.2 + 1.9 + 3.1$.

Exercise

Exercise 2

Give a run whose trace is $(a, \ldots), (b, \ldots), (a, \ldots), (b, \ldots)$.

Timed automaton: Definition

Definition

A timed automaton $\mathcal{A}$ is a tuple $(L, \ell^o, \Sigma, X, \longrightarrow, \text{Inv})$ in which:

- L is a finite set of locations;
- $\ell^o \in L$ is the initial location;
- Σ is an alphabet of actions;
- X is the set of clock variables or simply clocks (see later ...);
- $\longrightarrow \subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ is the set of edges (see later ...);
- $\text{Inv} : L \to \mathcal{C}(X)$ is the invariant mapping each location to a clock constraint (see later ...).

Explanations

The domain of a clock is $\mathbb{R}^+$. A clock measures time in a continuous way. Time advances implicitly. All the clocks are incremented synchronously.

Explanations

The domain of a clock is $\mathbb{R}^+$. A clock measures time in a continuous way. Time advances implicitly. All the clocks are incremented synchronously.

Definition (cont.)

$\longrightarrow$ is the "edge relation" defined by a set of quintuples of $L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$. A quintuple $(\ell_i, g, a, X', \ell_f)$ is read as follows:

- ℓ_i is the source location ℓ_f the target location of the edge;
- g is the guard, which is a clock constraint;
- a is the action label;
- $X' \subseteq X$ are the clocks to be reset when firing the edge. In the graphical representation we use assignments of the form $x = 0$ to indicate the clocks to be reset.

Exercise 3

The domain of a clock is $\mathbb{R}^+$. What does this mean?
What does the notation 2^X mean?

Explanations (2)

An invariant is a clock constraint associated with a location ℓ. It must hold while the automaton is in ℓ. The automaton must immediately exit ℓ just before the invariant turns false due to the passing of time.

Clock constraints

Clock constraints are generated by the following grammar:

$$\mathcal{C} ::= x \bowtie c \mid \mathcal{C} \wedge \mathcal{C}$$

where $\bowtie \in \{\leq, <, ==, >, \geq\}$ and c is an integer and x is a clock from a finite set of clocks X.

Remark: The disjunction of two constraints $(\mathcal{C} \vee \mathcal{C})$ or the negation of a constraint$(\neg\mathcal{C})$ are not allowed.

Exercise 4

What does the notation $::=$ and $\mid$ mean?
Can the restriction "$\mathcal{C} \vee \mathcal{C}$ forbidden" easily be circumvented?

Constraint examples

- $x \leq 5$
- $x \geq 3 \wedge y \leq 9$
- $x > 4 \wedge y == 10$
- $x < 4 \wedge y \leq 10$

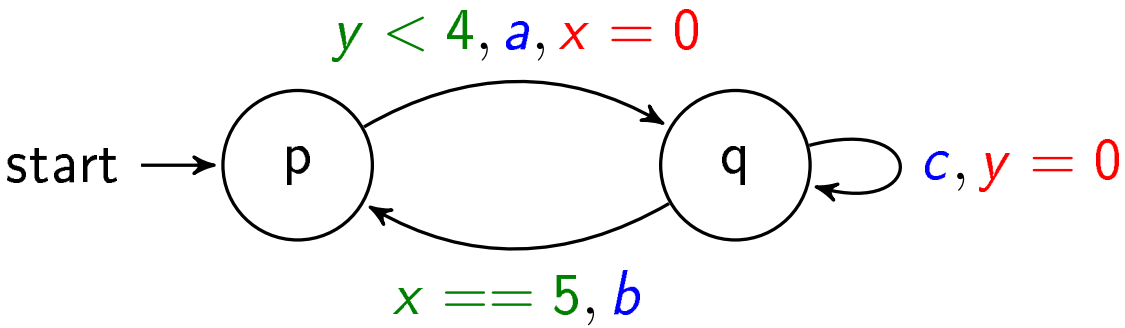
Questions:

Exercise 5

Forum!

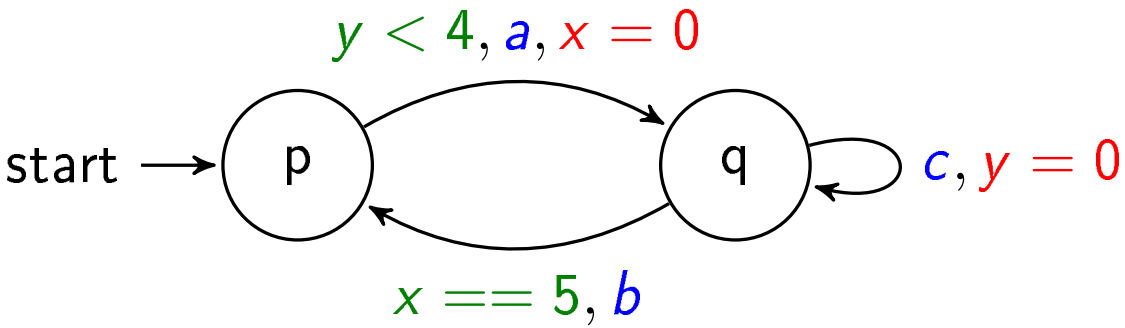
- Is $x \leq \pi$ a constraint?
- Can we write $x == 2$?
- Can we write $\neg(x == 2)$?

Timed automaton run (repeated)



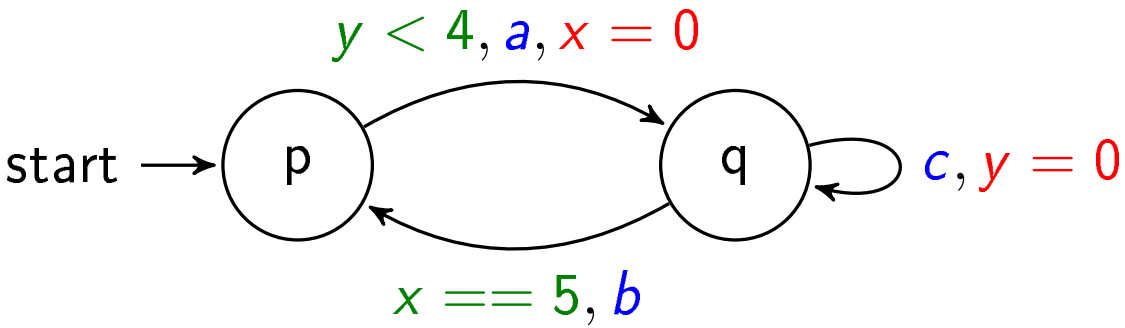
Defining a state slightly more formally:

Timed automaton run (repeated)



Defining a state slightly more formally:
A clock valuation is a function $\eta : X \to \mathbb{R}^+$.
A state is a pair (location, clock valuation).

Timed automaton run (repeated)



Defining a state slightly more formally:
A clock valuation is a function $\eta : X \to \mathbb{R}^+$.
A state is a pair (location, clock valuation).
A run is a sequence of alternating delay and discrete transitions:

$$\begin{pmatrix} p \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow[3.2]{delay} \begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix} \xrightarrow[a]{discrete} \begin{pmatrix} q \\ 0 \\ 3.2 \end{pmatrix} \xrightarrow[1.9]{delay} \begin{pmatrix} q \\ 1.9 \\ 5.1 \end{pmatrix} \xrightarrow[c]{discrete}$$

$$\begin{pmatrix} q \\ 1.9 \\ 0.0 \end{pmatrix} \xrightarrow[3.1]{delay} \begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix} \xrightarrow[b]{discrete} \begin{pmatrix} p \\ 5.0 \\ 3.1 \end{pmatrix}$$

Exercise 6

Explain the previous slide in some words:

- What is X in this example?
- How is, e.g., $\begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix}$ a pair (location, clock valuation)?

Timed automaton trace (repeated)

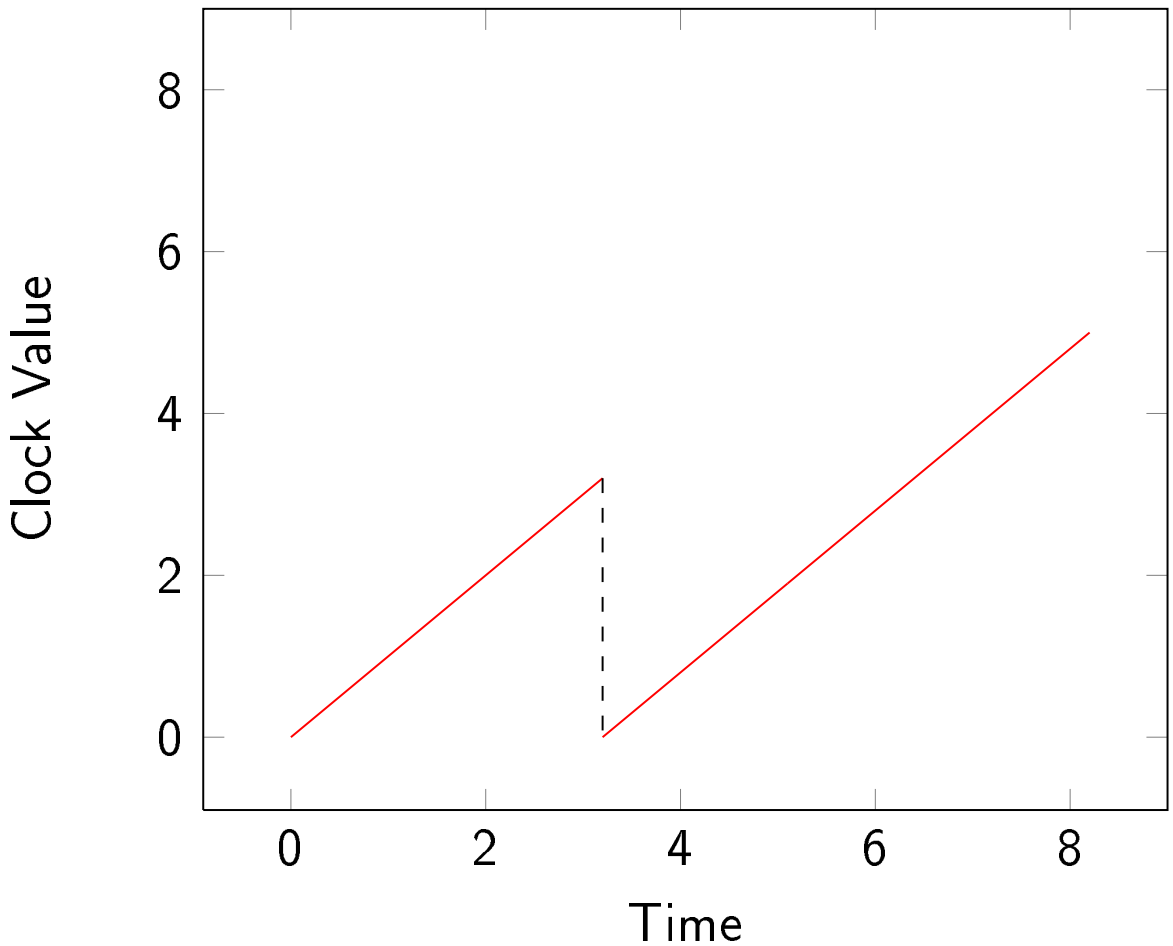
Given a run, e.g.

$$\begin{pmatrix} p \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow[3.2]{\textit{delay}} \begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix} \xrightarrow[a]{\textit{discrete}} \begin{pmatrix} q \\ 0 \\ 3.2 \end{pmatrix} \xrightarrow[1.9]{\textit{delay}} \begin{pmatrix} q \\ 1.9 \\ 5.1 \end{pmatrix} \xrightarrow[c]{\textit{discrete}}$$

$$\begin{pmatrix} q \\ 1.9 \\ 0.0 \end{pmatrix} \xrightarrow[3.1]{\textit{delay}} \begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix} \xrightarrow[b]{\textit{discrete}} \begin{pmatrix} p \\ 5.0 \\ 3.1 \end{pmatrix}$$

a trace simply collects all the actions plus their absolute occurrence times.
In this example: $(a, 3.2)(c, 5.1)(b, 8.2)$.
Each absolute occurrence time is simply the sum of all previous delays, e.g. $8.2 = 3.2 + 1.9 + 3.1$.

Clock evolution



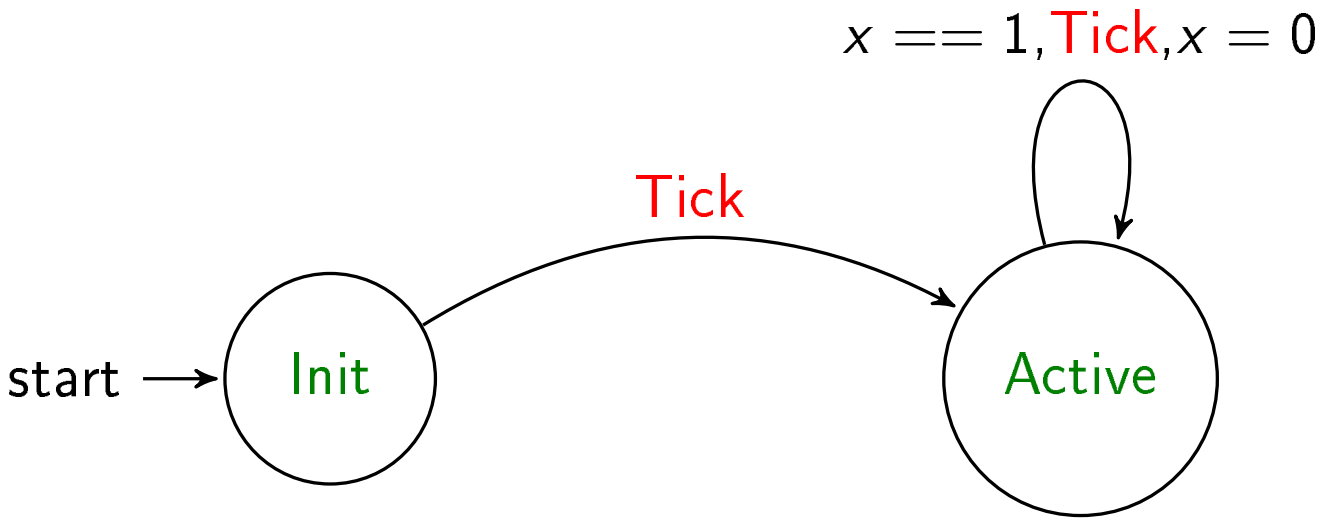
The clock evolves with time. It can be reset, but afterwards in continues to run immediately.

Motivation for invariants

We have not looked at location invariants yet!
Consider a device that goes "Tick" initially and then again every second.

$X = \{x\}$

S1:

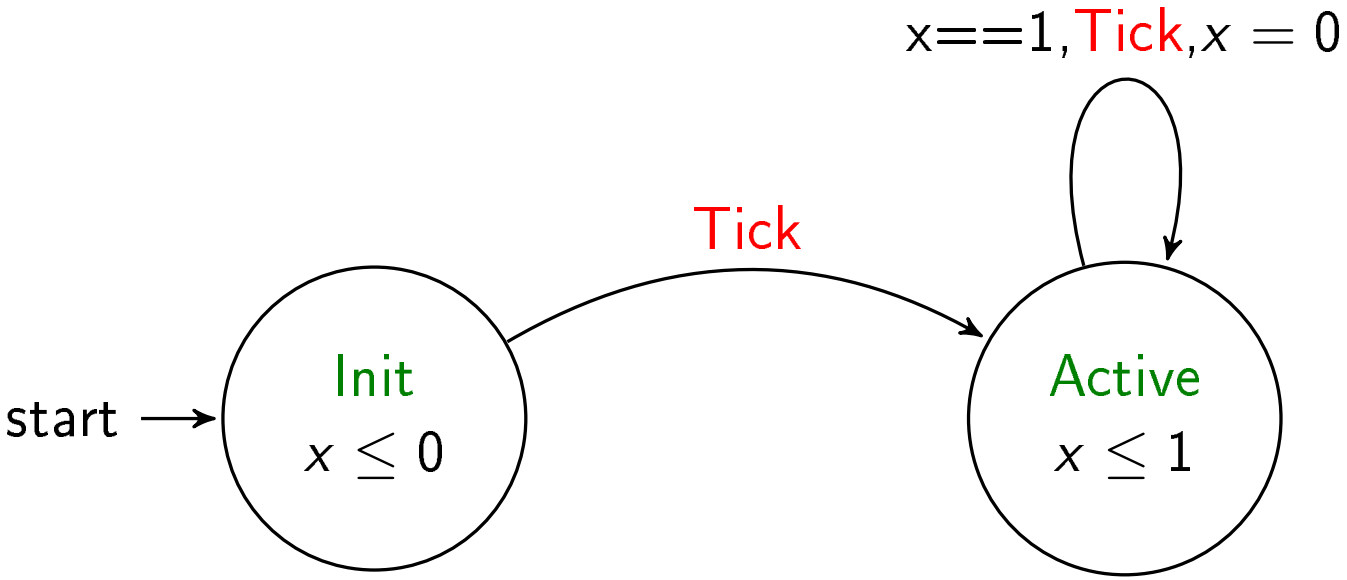


Exercise 7

Does this timed automaton enforce the behaviour described above?

Motivation for invariants (2)

S2:



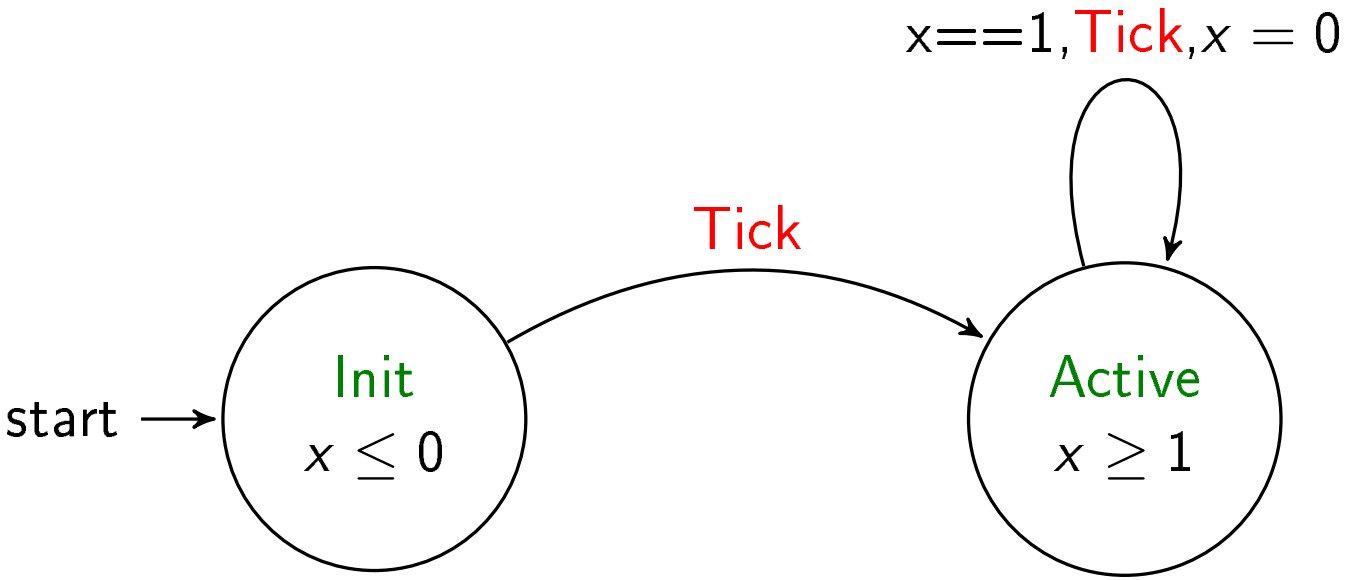
The invariant forces the automaton to stay in Active for at most one second before taking the edge going "Tick".

Exercise 8

What are the traces/runs of this automaton?

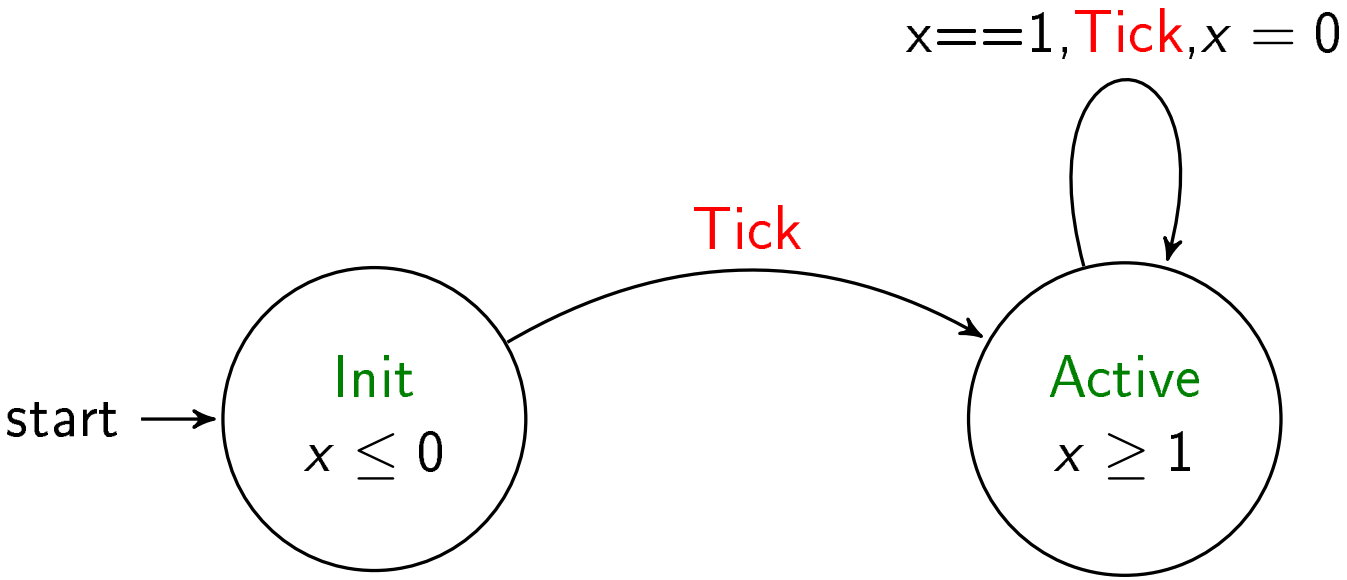
A strange invariant

S2:



A strange invariant

S2:



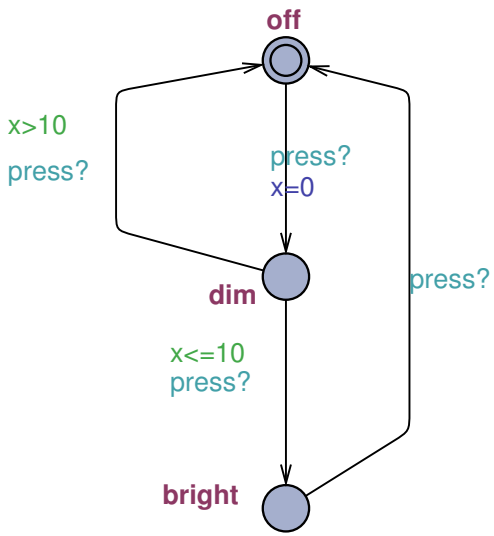
Following [BY04], we require that invariants have the form $x < c$ or $x \leq c$ (downwards-closed invariants).

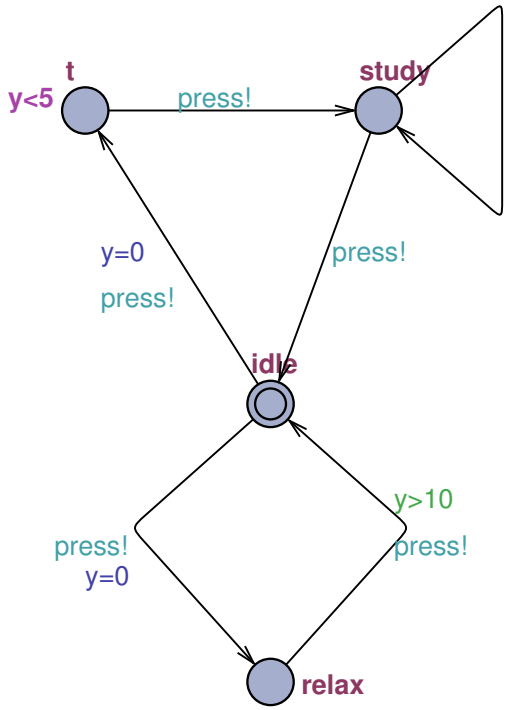
Reason: Invariants must be met when the location is entered and are there to say when it is time to exit the location. So invariants must have a form that ensures that they become false as time passes, not become true.

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

An example





This is a model for a lamp that can be dimmed ("simple click") or bright ("double click"), and its user.

Exercise 9

What is the difference between "double clicking" and "clicking twice"?

The alphabet

For the purpose of composition, we consider several component automata that share the clock set X and the alphabet Σ.
Σ contains a special symbol (action) τ which is used whenever a component does a transition privately.
The symbols in $\Sigma \setminus \{\tau\}$ are called channels.
We assume that in each component automaton, each edge is labelled either by τ, or by (send action) $c!$ or (receive action) $c?$, where $c \in \Sigma \setminus \{\tau\}$. For a given $c \in \Sigma \setminus \{\tau\}$, we say that $c!$ and $c?$ are matching actions.

Definition of product (composition)

We consider n timed automata $\mathcal{A}_i = (L_i, \ell_i^o, \Sigma, X, \longrightarrow_i, \mathsf{Inv}_i)$, $i = 1, \ldots, n$.
The product $\mathcal{A}_1 \| \ldots \| \mathcal{A}_n$ is the automaton $(L, \ell^o, \Sigma, X, \longrightarrow, \mathsf{Inv})$ where:

- $L = L_1 \times \ldots \times L_n$;
- $\ell^0 = (\ell_1^0, \ldots, \ell_n^0)$;
- $\longrightarrow$ is defined as:
  - private edge: for all $(\ell_1, \ldots, \ell_n) \in L$, if for some $i \in \{1, \ldots, n\}$, we have if $(\ell_i, g_i, \tau, r_i, \ell_i') \in \longrightarrow_i$, then $((\ell_1, \ldots, \ell_i, \ldots, \ell_n), g_i, \tau, r_i, (\ell_1, \ldots, \ell_i', \ldots, \ell_n)) \in \longrightarrow$;

Definition of product (composition)

We consider n timed automata $\mathcal{A}_i = (L_i, \ell_i^o, \Sigma, X, \longrightarrow_i, \mathsf{Inv}_i)$, $i = 1, \ldots, n$.
The product $\mathcal{A}_1 \| \ldots \| \mathcal{A}_n$ is the automaton $(L, \ell^o, \Sigma, X, \longrightarrow, \mathsf{Inv})$ where:

- $L = L_1 \times \ldots \times L_n$;
- $\ell^0 = (\ell_1^0, \ldots, \ell_n^0)$;
- $\longrightarrow$ is defined as:
  - private edge: for all $(\ell_1, \ldots, \ell_n) \in L$, if for some $i \in \{1, \ldots, n\}$, we have if $(\ell_i, g_i, \tau, r_i, \ell_i') \in \longrightarrow_i$, then $((\ell_1, \ldots, \ell_i, \ldots, \ell_n), g_i, \tau, r_i, (\ell_1, \ldots, \ell_i', \ldots, \ell_n)) \in \longrightarrow$;
  - synchronised edge: for all $(\ell_1, \ldots, \ell_n) \in L$, if for some $i, j \in \{1, \ldots, n\}$, $i < j$, we have $(\ell_i, g_i, c', r_i, \ell_i') \in \longrightarrow_i$ and $(\ell_j, g_j, c'', r_j, \ell_j') \in \longrightarrow_j$ where c'' and c' are matching actions, then $((\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n), g_i \wedge g_j, c, r_i \cup r_j, (\ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n)) \in \longrightarrow$.
- $\mathsf{Inv}((\ell_1, \ldots, \ell_n)) = Inv_1(\ell_1) \wedge \ldots \wedge Inv_n(\ell_n)$;

Exercise 10

Why does it say $\mathcal{A}_i = (L_i, \ell_i^o, \Sigma, X, \longrightarrow_i, \mathsf{Inv}_i)$ and not $\mathcal{A}_i = (L_i, \ell_i^o, \Sigma_i, X_i, \longrightarrow_i, \mathsf{Inv}_i)$

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

Uppaal

url: www.uppaal.com
Mature tool!

- Graphical editor
- Simulator
- Verifier (model checker)

Guards in Uppaal

Recall the definition of clock constraints:

$$\mathcal{C} ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid x == c \mid \mathcal{C} \wedge \mathcal{C}$$

In Uppaal we do not necessarily have to put a concrete number for a constant c, but we can declare an integer constant parametrically (for easy maintenance).

Example of synchronisation

Declaration of channel:

```
chan c, d, e;
```

Usage:

- send

  c!

- receive

  c?

The two edges of the two processes synchronise via the channel. One process is the sender and the other the receiver.

Exercise 11

What is a **process**? Compare to the notions of the previous section.

Templates

- To save effort and reduce a source of possible errors, there is special support in Uppaal for defining processes that are identical up to some constants. In Uppaal, a process is an instantiation of a template.
- The template contains one or more parameters that can be instantiated.

Example:

- Template declaration:

```
process semaphore(int n)
...
```

- Process declaration:

```
aMutexSem1 = semaphore(1);
aMutexSem2 = semaphore(2);
```

- System declaration:

```
system aMutexSem1 aMutexSem2;
```

Temporal logic used by Uppaal

So far, we can use Uppaal to model the actual behaviour of systems. What is ultimately needed is to verify that the actual behaviour corresponds to desired behaviour such as (recall lecture on model checking):

- If a process asks infinitely often for being executed then the operating system will eventually execute it;
- It is always possible to get back to the initial state;
- ...

Temporal logic used by Uppaal

So far, we can use Uppaal to model the actual behaviour of systems.
What is ultimately needed is to verify that the actual behaviour corresponds to desired behaviour such as (recall lecture on model checking):

- If a process asks infinitely often for being executed then the operating system will eventually execute it;
- It is always possible to get back to the initial state;
- ...

To express such desired behaviours, Uppaal uses the temporal logic CTL.
We are brief here ...

CTL subset

The language contains

- atomic state formulas:
  - $x \leq 3$, $y \leq 5$, $i == 10$ ...
  - The formula $P.\ell$ expresses that the process P is in location ℓ.
  - The formula deadlock expresses that no transition is possible.
- Boolean combinations

$$p ::= s \mid p \textbf{ and } p \mid p \textbf{ or } p \mid \textbf{not } p \mid p \textbf{ imply } p \mid (p)$$

  where s is an atomic state formula.
- path formulas built using exactly one path quantifier:
  - E<> p: there exists a run such that for some state of this run, p holds,
  - A[] p: for all runs, for all states of each run, p holds.

Exercise 12

Write down 4 different CTL formulas each involving at least one Boolean operator and one path quantifier.

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

Semantics

We will now formally define the runs of a timed automaton. The semantics of a timed automaton is expressed as a transition system.
Recall that a state is a pair (location, clock valuation) ...

Definition

Definition

Let $\mathcal{A} = (L, \ell^o, \Sigma, X, \longrightarrow, \mathsf{Inv})$. We define its semantics as the transition system: $S(\mathcal{A}) = (S, \Sigma', \longrightarrow, I)$ where

- $S = L \times (X \longrightarrow \mathbb{R}^+)$ (the state set)
- $\Sigma' = \Sigma \cup \mathbb{R}^+$ (the alphabet)
- $I = \{(\ell^0, \eta) \mid \forall x \in X.\ \eta(x) = 0\}$ (initial state)
- The transition relation $\longrightarrow$ is defined through two rules:
  - discrete transition: $(\ell, \eta) \xrightarrow{a} (\ell', \eta')$ if
    - the timed automaton has an edge (ℓ, g, a, X', ℓ')
    - $\eta \models g$
    - $\eta' = [X' = 0]\eta$
    - $\eta' \models \mathsf{Inv}(\ell')$
  - delay transition: $(\ell, \eta) \xrightarrow{\delta} (\ell, \eta + \delta)$ with $\delta \in \mathbb{R}^+$ if $\forall d : 0 \leq d \leq \delta \Rightarrow \eta + d \models \mathsf{Inv}(\ell)$.

Note: there is an infinite number of states and transitions.

Explanations

The entailment $\models$ is defined by interpreting $<, \leq, \ldots, \wedge$ in the standard way, e.g.:

- $\{x \mapsto 0.5, y \mapsto 1.0\} \models y > 0.0 \wedge x < 1.0$
- $\{x \mapsto 0.5, y \mapsto 1.0\} \not\models y > 0.0 \wedge x > 3.0$

Explanations

The entailment $\models$ is defined by interpreting $<, \leq, \ldots, \wedge$ in the standard way, e.g.:

- $\{x \mapsto 0.5, y \mapsto 1.0\} \models y > 0.0 \wedge x < 1.0$
- $\{x \mapsto 0.5, y \mapsto 1.0\} \not\models y > 0.0 \wedge x > 3.0$

$[X' = 0]\eta$ is defined as

$$([X' = 0]\eta)(x) := \left\{ \begin{array}{ll} 0 & \text{if } x \in X' \\ \eta(x) & \text{if } x \notin X' \end{array} \right\}$$

i.e., setting all clocks in X' to 0 and leaving the other clocks unchanged.

Explanations

The entailment $\models$ is defined by interpreting $<, \leq, \ldots, \wedge$ in the standard way, e.g.:

- $\{x \mapsto 0.5, y \mapsto 1.0\} \models y > 0.0 \wedge x < 1.0$
- $\{x \mapsto 0.5, y \mapsto 1.0\} \not\models y > 0.0 \wedge x > 3.0$

$[X' = 0]\eta$ is defined as

$$([X' = 0]\eta)(x) := \left\{ \begin{array}{ll} 0 & \text{if } x \in X' \\ \eta(x) & \text{if } x \notin X' \end{array} \right\}$$

i.e., setting all clocks in X' to 0 and leaving the other clocks unchanged. The clock valuation $\eta + d$ is defined as

$$(\eta + d)(x) := \eta(x) + d \quad \text{for all clocks } x,$$

i.e., all clocks are advanced by d.

Examples of these definitions

Exercise 13

1. Does $\{x \mapsto 1.0, y \mapsto 1.0\} \models y > 0.0 \wedge x < 1.0$ hold?
2. Does $\{x \mapsto 1.0, y \mapsto 1.0\} \models y > 0.0 \wedge x \leq 1.0$ hold?
3. Does $\{x \mapsto 1.0, y \mapsto 2.0\} \models x > 1.0$ hold?
4. $[\{x\} = 0]\{x \mapsto 1.0, y \mapsto 1.0\} = \ldots?$
5. $[\{x, y\} = 0]\{x \mapsto 1.0, y \mapsto 1.0\} = \ldots?$
6. $[\{x\} = 0]\{x \mapsto 0.0, y \mapsto 1.0\} = \ldots?$
7. $\{x \mapsto 1.0, y \mapsto 1.0\} + 0.5 = \ldots?$
8. $\{x \mapsto 1.0, y \mapsto 2.0\} + 0.5 = \ldots?$

Timed automaton run (repeated)

A run starting from a state is a finite or infinite sequence of alternating delay and discrete transitions.

$$\begin{pmatrix} p \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow[3.2]{\textit{delay}} \begin{pmatrix} p \\ 3.2 \\ 3.2 \end{pmatrix} \xrightarrow[a]{\textit{discrete}} \begin{pmatrix} q \\ 0 \\ 3.2 \end{pmatrix} \xrightarrow[1.9]{\textit{delay}} \begin{pmatrix} q \\ 1.9 \\ 5.1 \end{pmatrix} \xrightarrow[c]{\textit{discrete}}$$

$$\begin{pmatrix} q \\ 1.9 \\ 0.0 \end{pmatrix} \xrightarrow[3.1]{\textit{delay}} \begin{pmatrix} q \\ 5.0 \\ 3.1 \end{pmatrix} \xrightarrow[b]{\textit{discrete}} \begin{pmatrix} p \\ 5.0 \\ 3.1 \end{pmatrix}$$

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

Region graph

A state of the timed transition system of a timed automaton is a couple:

$$(\ell, \eta).$$

Exercise 14

What is ℓ? What is η?

The state space as well as the branching of this transition system is infinite.

Region graph

A state of the timed transition system of a timed automaton is a couple:

$$(\ell, \eta).$$

Exercise 14

What is ℓ? What is η?

The state space as well as the branching of this transition system is infinite. The algorithmic verification of timed automaton properties is possible thanks to the region graph technique by Alur and Dill [AD94] ([BY04]). The reasoning on the infinite state space is replaced by a reasoning on a finite partition of the state space. An element of this partition is called a region. All elements of the region have the same relevant properties:

- same discrete transitions;
- same delay transitiitions.

Regions: the intuition

Even though there are infinitely many clock valuations, what matters really?

- For knowing whether a discrete transition can be taken or not, it may matter whether for some clock x, it holds that $x < c$, $x = c$, or $x > c$.

Regions: the intuition

Even though there are infinitely many clock valuations, what matters really?

- For knowing whether a discrete transition can be taken or not, it may matter whether for some clock x, it holds that $x < c$, $x = c$, or $x > c$.
- For knowing which clock x, among all the clocks, will be the next one to change its value (due to the passing of time)
  - from $x < c$ to $x = c$; or
  - from $x = c$ to $x > c$; or
  - from $x > c$ to $x = c + 1$,

  the ordering of the fractional parts of the clocks matters.

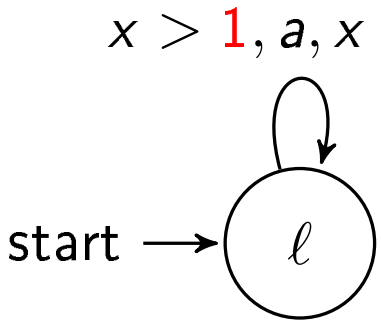
Thus, some assigments must be distinguished whereas others can be considered as equivalent, for our purposes.
Regions capture exactly this information.

Exercise 15

Why is it reasonable to say that there is definitely no essential difference between $\{x \mapsto 1.5, y \mapsto 3.3\}$ and $\{x \mapsto 1.6, y \mapsto 3.4\}$?
Why might there be an essential difference between $\{x \mapsto 1.5, y \mapsto 3.3\}$ and $\{x \mapsto 2.0, y \mapsto 3.8\}$?

Region automaton example

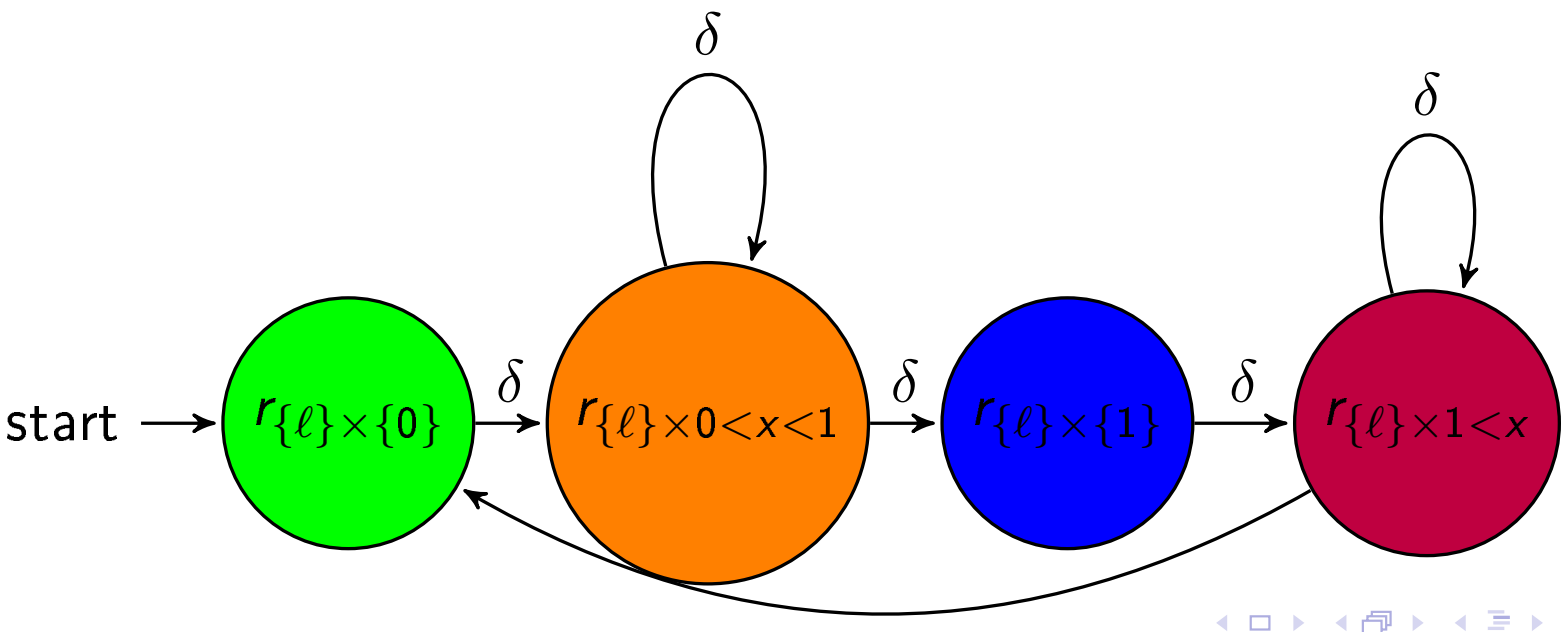
Consider the following automaton:



We partition its state space as follows :

$$\{\{(\ell, x) \mid x = 0\}, \{(\ell, x) \mid 0 < x < 1\}, \{(\ell, x) \mid x = 1\}, \{(\ell, x) \mid 1 < x\}\}$$

Then its region graph is the following:



Properties of the region graph

- Finite number of states.
- Finite number of transitions (finite branching).

The "equivalence" between the region graph automaton and the transitition system of a timed automaton allows us to decide basic temporal properties over timed automata.

Region equivalence

Some preliminaries:

- For any clock variable x, let C_x be the largest integer appearing in constraints involving x.
- for $t \in \mathbb{R}$, its integral part is denoted: $\lfloor t \rfloor$, its fractional part is denoted fract(t)

$$\lfloor 2.32 \rfloor = 2 \quad \text{fract}(2.32) = 0.32$$

Clock valuation equivalence

Example: We have a timed automaton with two clocks x and y. x is compared to 1 and 2: $C_x = 2$. y is compared to 0 and 1: $C_y = 1$. The equivalence classes are:

- Corner points: $(0,0), (1,1)$,
- open line segments $\{(x,y) : (0 < x < 1) \wedge (x = y)\}$,
- open regions
  $\{(x,y) : 0 < x < y < 1\}, \{(x,y) : (1 < x < 2) \wedge (y > 1)\}$
- ...

Region equivalence relation

Visual illustration later ...

Definition ($\equiv_{\mathsf{REG}}$)

Two valuations η and η' are region-equivalent: $\eta \equiv_{\mathsf{REG}} \eta'$ iff

- for all x, either $\lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$, or both $\eta(x) > C_x$ and $\eta'(x) > C_x$;
- for all x with $\eta(x) \leq C_x$, we have $\mathsf{fract}(\eta(x)) = 0$ iff $\mathsf{fract}(\eta'(x)) = 0$;
- for all x, y with $\eta(x) \leq C_x$ and $\eta(y) \leq C_y$, we have $\mathsf{fract}(\eta(x)) \leq \mathsf{fract}(\eta(y))$ iff $\mathsf{fract}(\eta'(x)) \leq \mathsf{fract}(\eta'(y))$.

Region equivalence relation

Visual illustration later ...

Definition ($\equiv_{\mathsf{REG}}$)

Two valuations η and η' are region-equivalent: $\eta \equiv_{\mathsf{REG}} \eta'$ iff

- for all x, either $\lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$, or both $\eta(x) > C_x$ and $\eta'(x) > C_x$;
- for all x with $\eta(x) \leq C_x$, we have $\mathsf{fract}(\eta(x)) = 0$ iff $\mathsf{fract}(\eta'(x)) = 0$;
- for all x, y with $\eta(x) \leq C_x$ and $\eta(y) \leq C_y$, we have $\mathsf{fract}(\eta(x)) \leq \mathsf{fract}(\eta(y))$ iff $\mathsf{fract}(\eta'(x)) \leq \mathsf{fract}(\eta'(y))$.

Given a valuation η, the set of all valuations η' such that $\eta \equiv_{\mathsf{REG}} \eta'$ is called the region of η, written $R(\eta)$.

Region equivalence relation

Visual illustration later ...

Definition ($\equiv_{\mathsf{REG}}$)

Two valuations η and η' are region-equivalent: $\eta \equiv_{\mathsf{REG}} \eta'$ iff

- for all x, either $\lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$, or both $\eta(x) > C_x$ and $\eta'(x) > C_x$;
- for all x with $\eta(x) \leq C_x$, we have $\mathsf{fract}(\eta(x)) = 0$ iff $\mathsf{fract}(\eta'(x)) = 0$;
- for all x, y with $\eta(x) \leq C_x$ and $\eta(y) \leq C_y$, we have $\mathsf{fract}(\eta(x)) \leq \mathsf{fract}(\eta(y))$ iff $\mathsf{fract}(\eta'(x)) \leq \mathsf{fract}(\eta'(y))$.

Given a valuation η, the set of all valuations η' such that $\eta \equiv_{\mathsf{REG}} \eta'$ is called the region of η, written $R(\eta)$.

- The number of regions is huge!
- The number of regions finite!
- Regions can be visualised geometrically ...

Region equivalence

Exercise 16

Let $C_x = 2, C_y = 3, C_z = 5$.

Determine which pairs of clock valuations are region-equivalent:

$1 : \{x \mapsto 1.3, y \mapsto 2.7, z \mapsto 4.4\}$ $2 : \{x \mapsto 1.4, y \mapsto 2.7, z \mapsto 4.3\}$

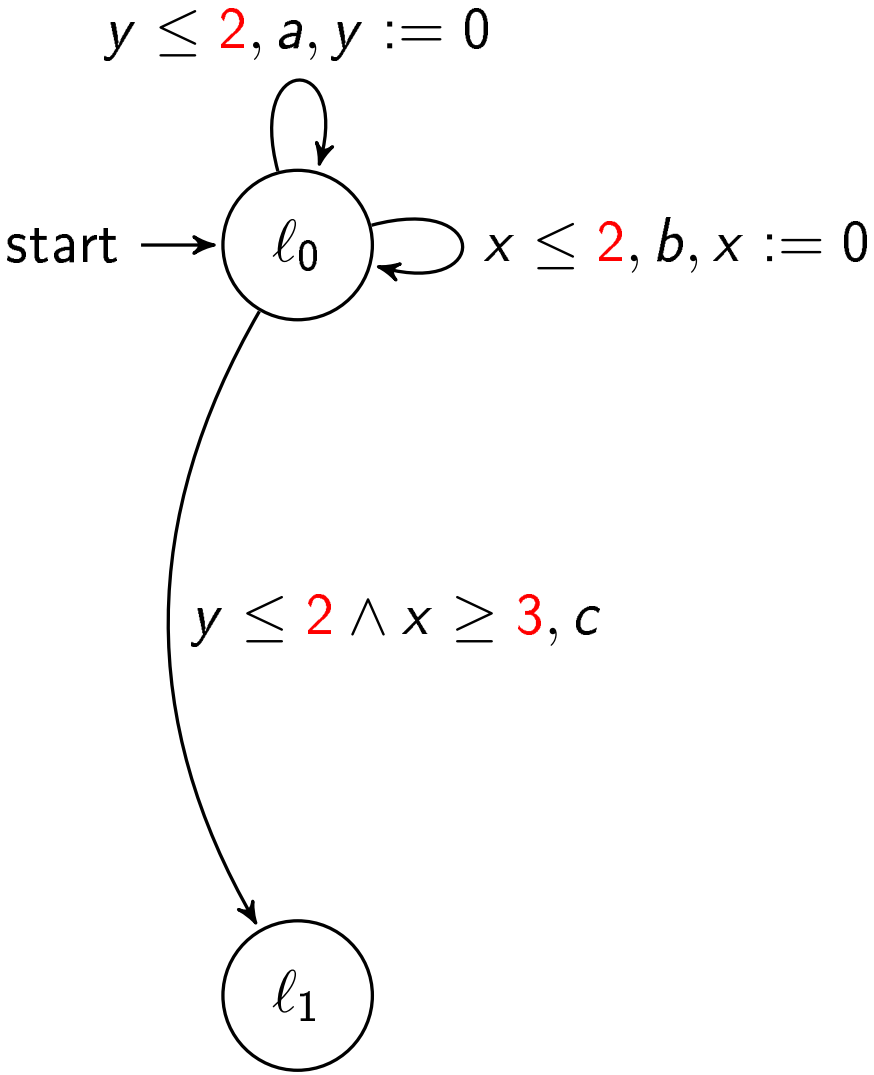
$3 : \{x \mapsto 1.3, y \mapsto 2.7, z \mapsto 5.5\}$ $4 : \{x \mapsto 1.4, y \mapsto 2.8, z \mapsto 4.5\}$

$5 : \{x \mapsto 1.3, y \mapsto 2.8, z \mapsto 4.4\}$ $6 : \{x \mapsto 1.3, y \mapsto 2.7, z \mapsto 10.0\}$

$7 : \{x \mapsto 1.7, y \mapsto 2.3, z \mapsto 10.0\}$ $8 : \{x \mapsto 1.3, y \mapsto 2.7, z \mapsto 4.5\}$

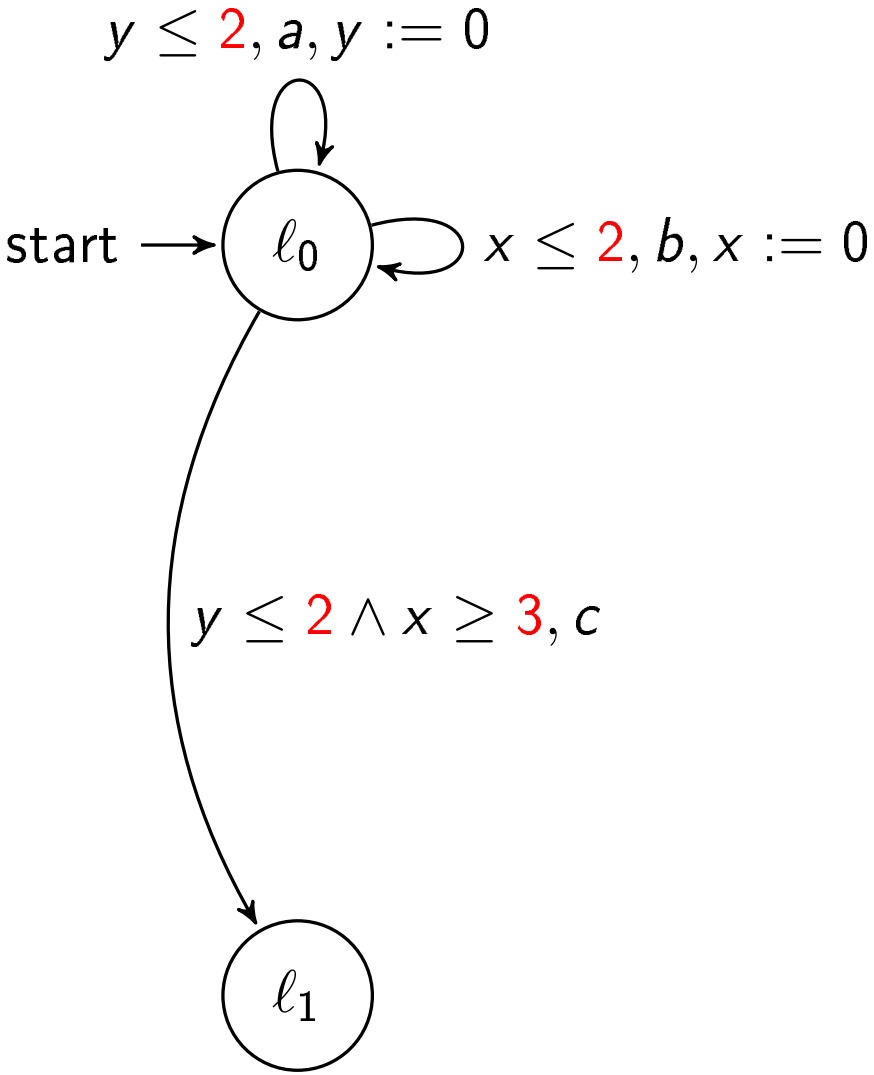
Region automaton example

Let us consider the following automaton A with the set of clocks $= \{x, y\}$.



Region automaton example

Let us consider the following automaton A with the set of clocks $= \{x, y\}$.



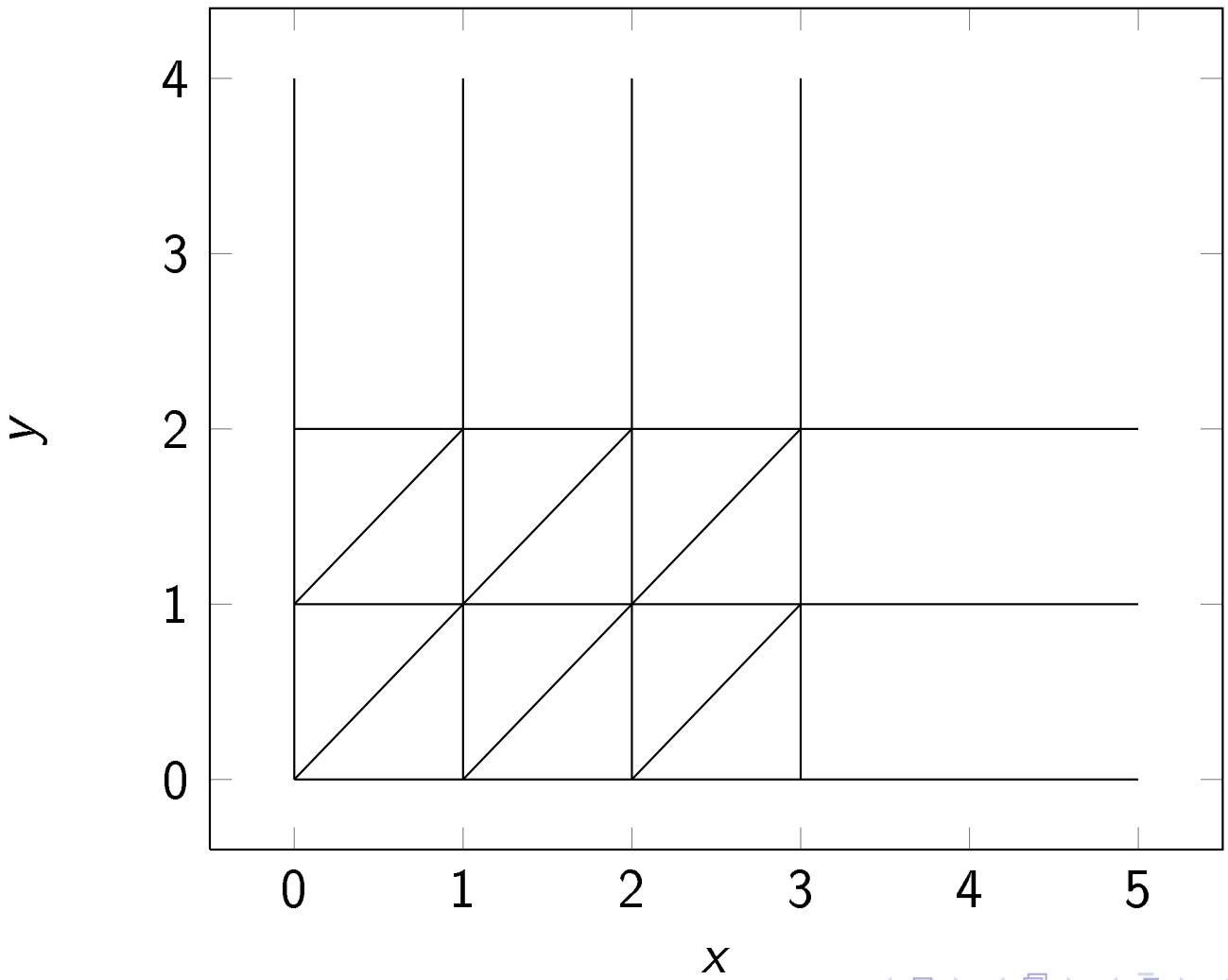
$C_x = 3$
$C_y = 2$

The regions of this example

$$\text{Regions}_{x,y} = \left\{ \begin{array}{ll} \{(x,y) \mid x = 0 \wedge y = 0\}, & \\ \vdots & \text{(points)} \\ \{(x,y) \mid x = 3 \wedge y = 2\}, & \\ \hline \{(x,y) \mid 0 < x \wedge x < 1 \wedge y = 0\}, & \\ \vdots & \text{(bounded segments)} \\ \{(x,y) \mid x = 3 \wedge 1 < y \wedge y < 2\}, & \\ \hline \{(x,y) \mid 3 < x \wedge y = 0\}, & \\ \vdots & \text{(unbounded segments)} \\ \{(x,y) \mid x = 3 \wedge y > 0\}, & \\ \hline \{(x,y) \mid 0 < x < 1 \wedge 0 < y < 1 \wedge x - y < 0\}, & \\ \vdots & \text{(bounded regions)} \\ \{(x,y) \mid 2 < x < 3 \wedge 1 < y < 2 \wedge x - y > 1\}, & \\ \hline \{(x,y) \mid 3 < x \wedge y < 1\}, & \\ \vdots & \text{(unbounded regions)} \\ \{(x,y) \mid 3 < x \wedge 2 < y\} & \end{array} \right\}$$

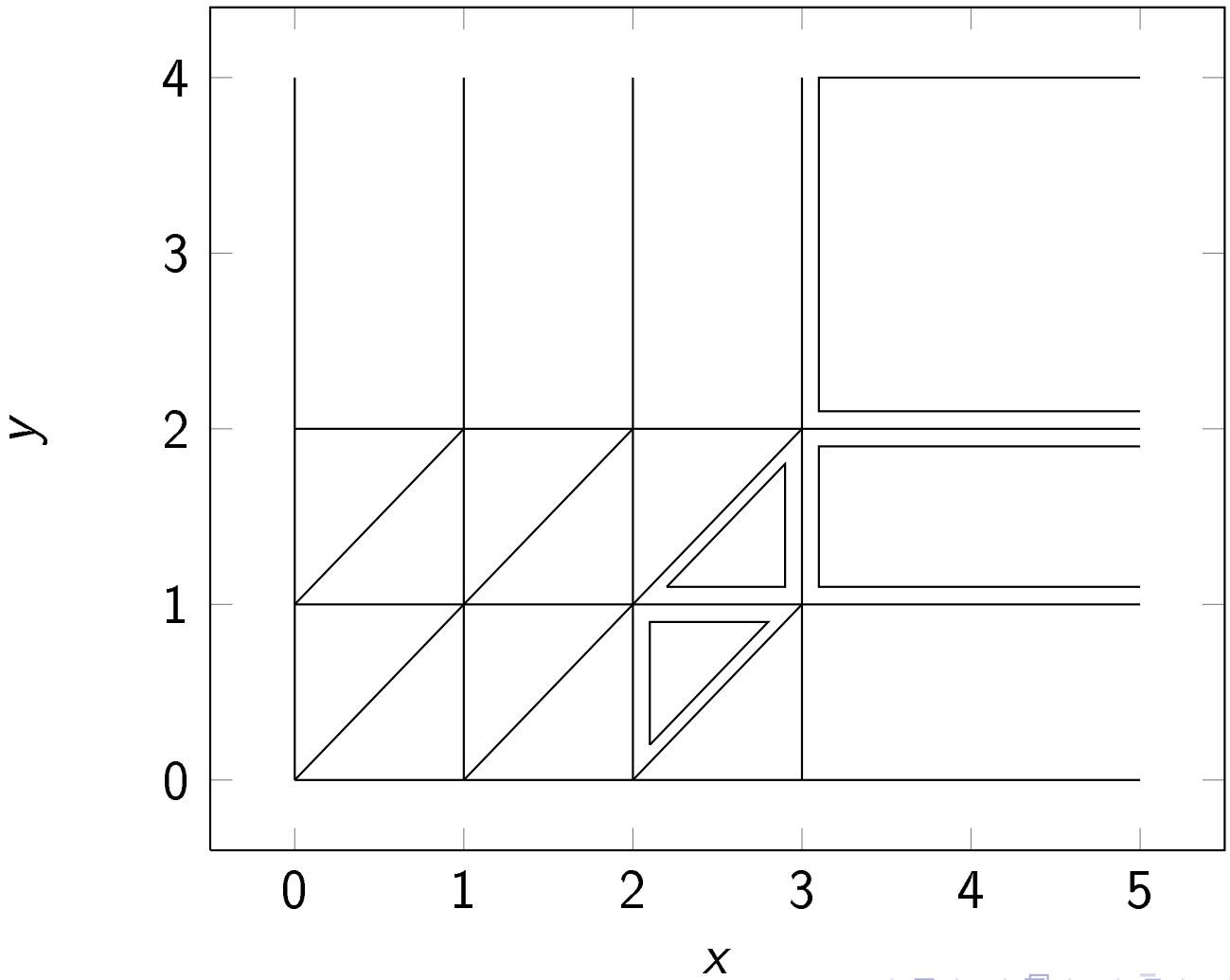
The regions of this example visualised

To simplify, we ignore discrete transitions and resets here. We only care for the passing of time.



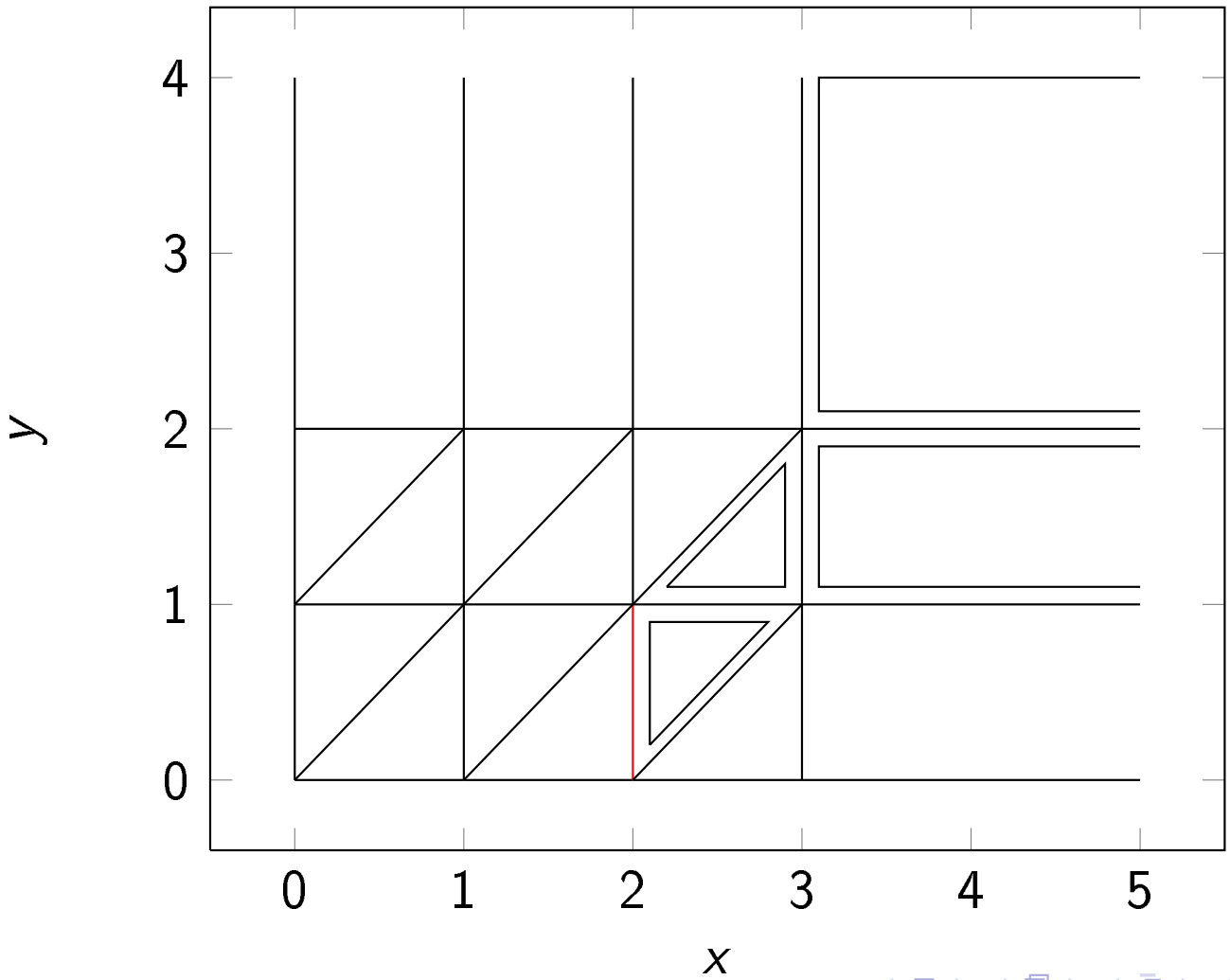
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



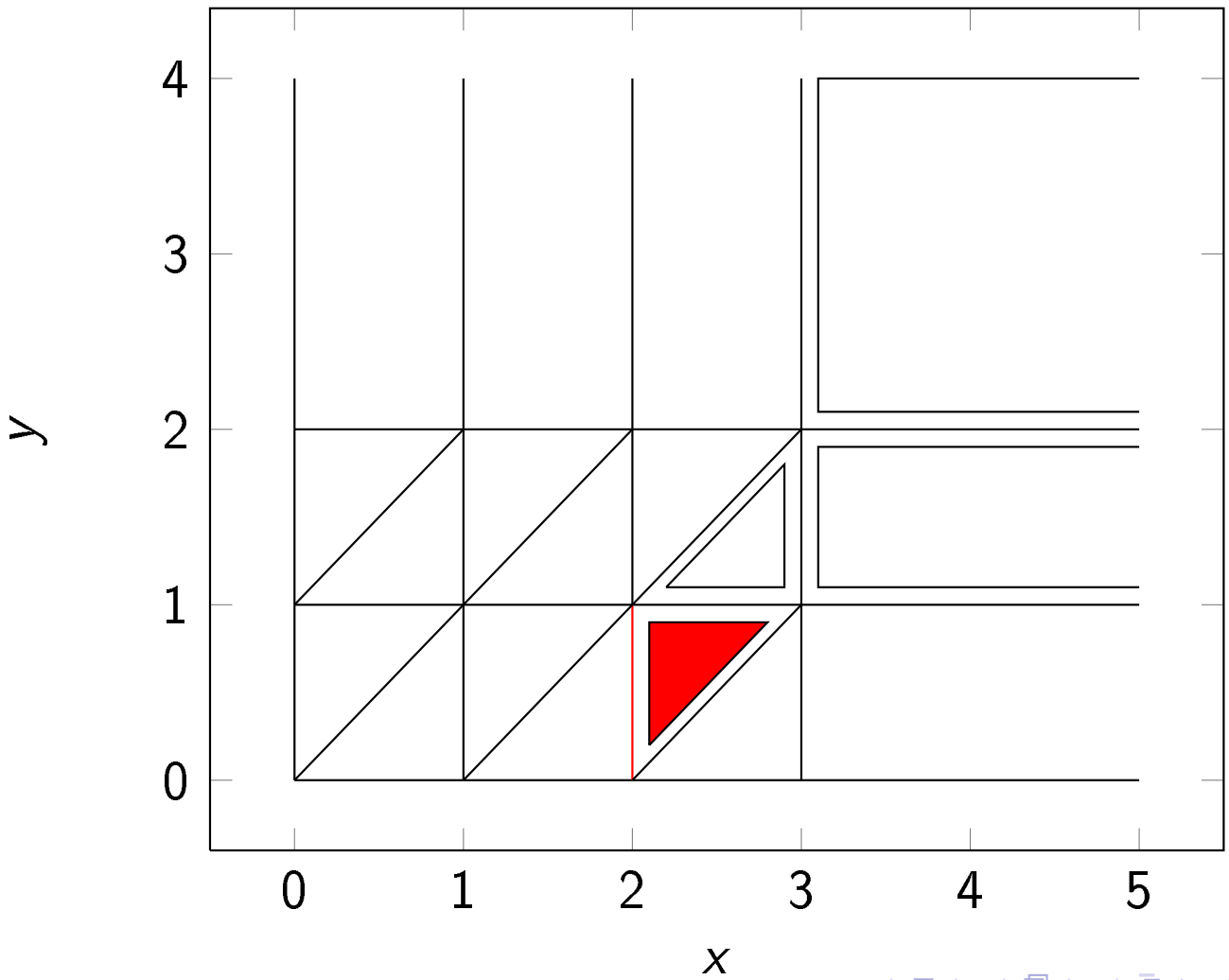
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



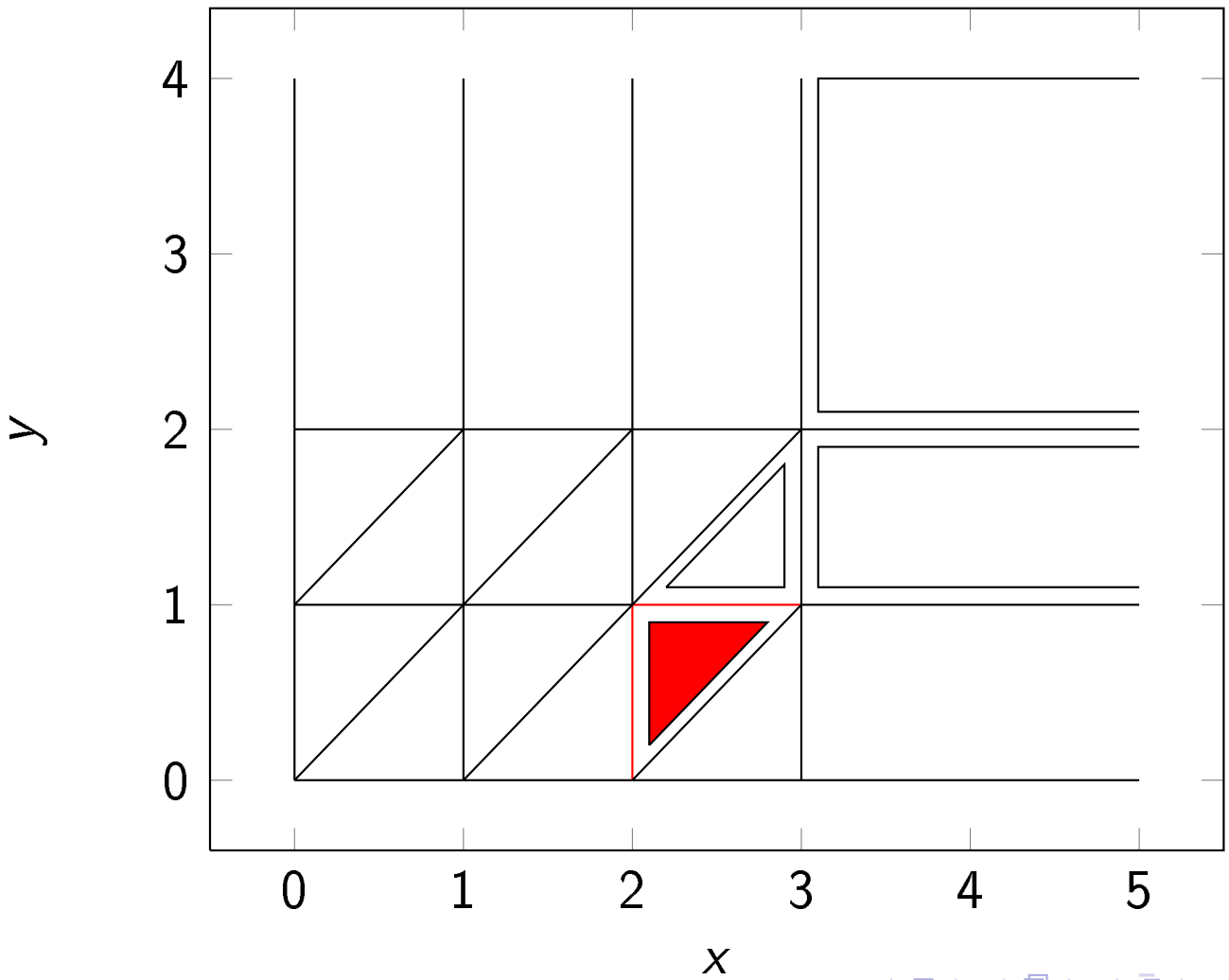
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



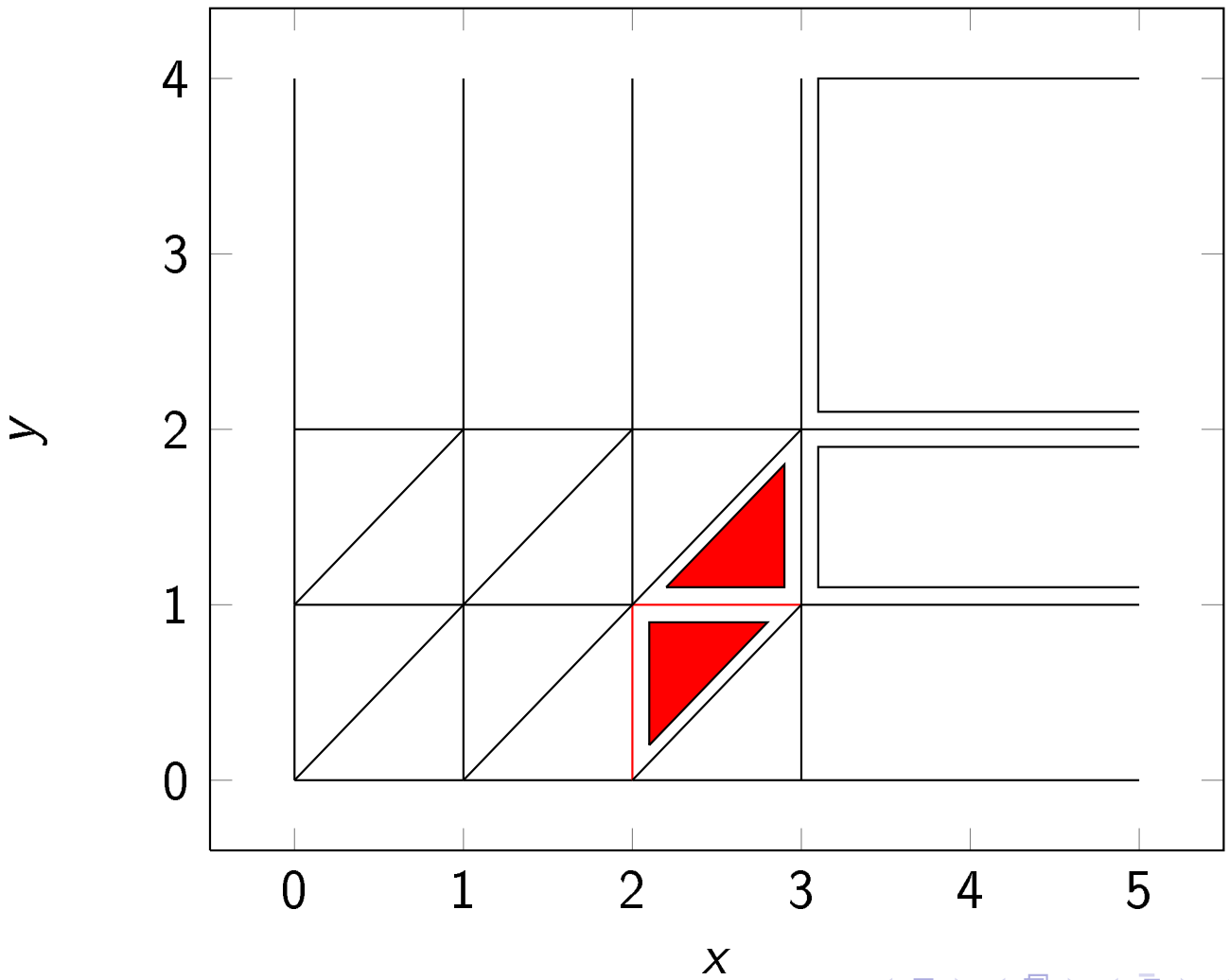
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



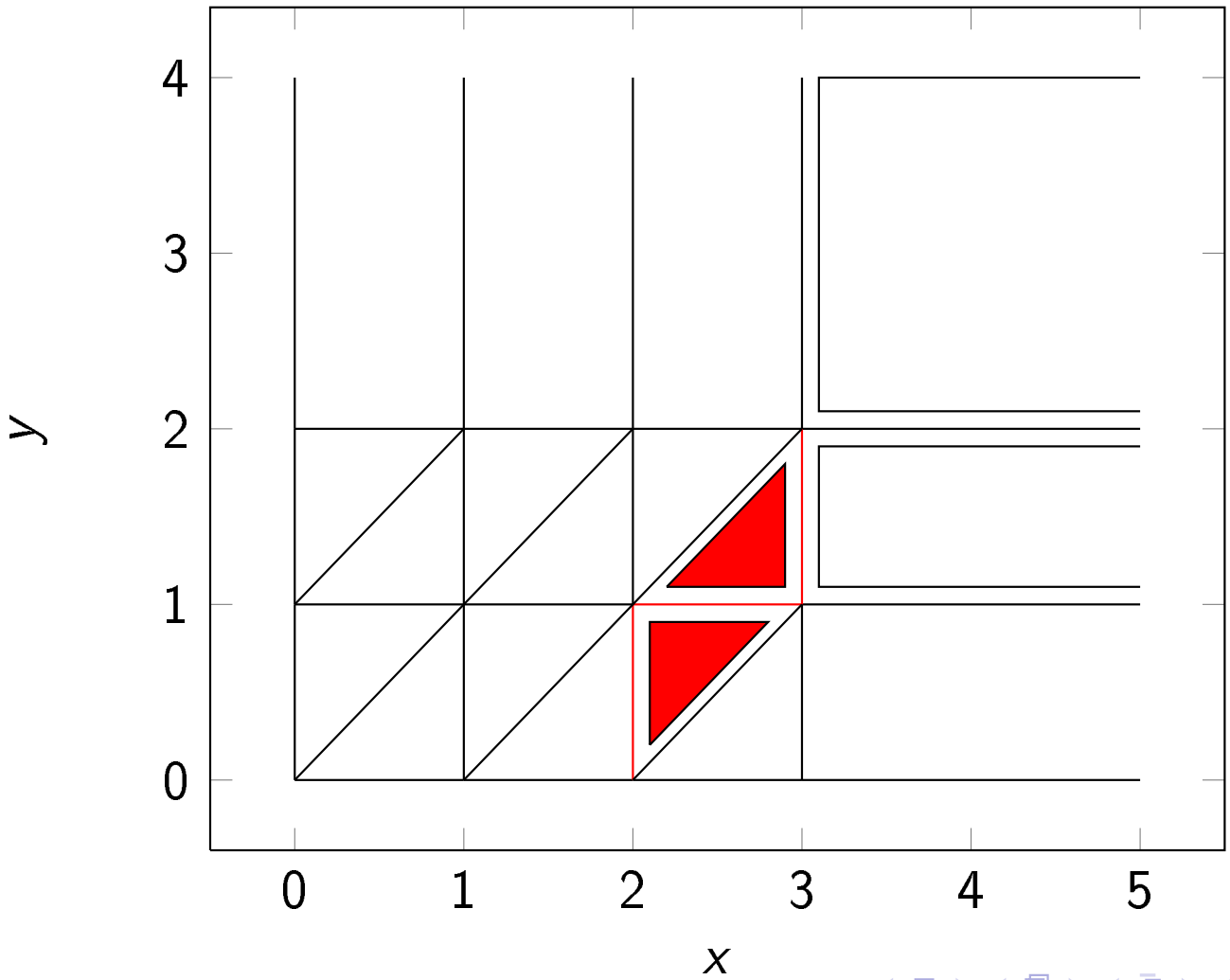
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



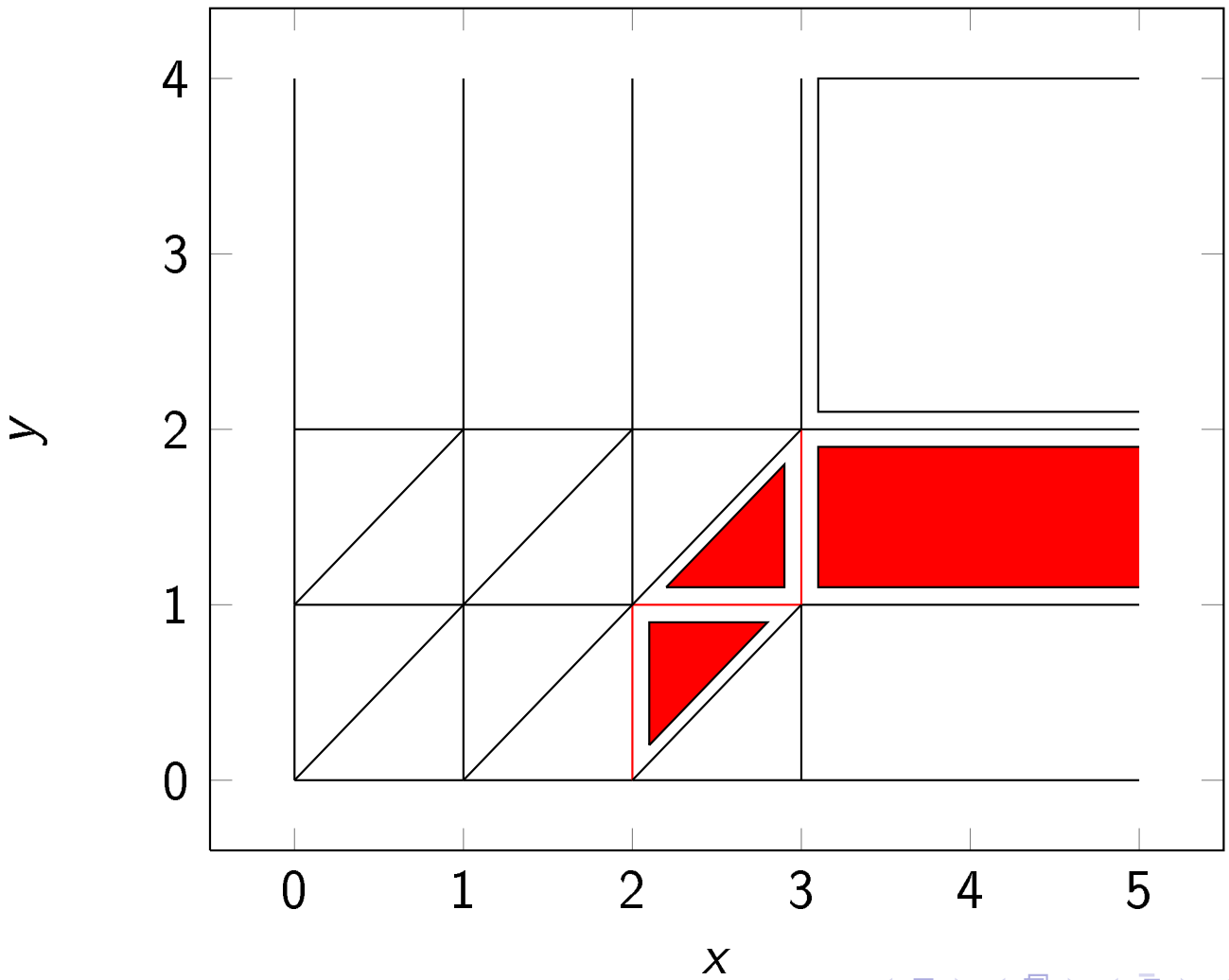
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



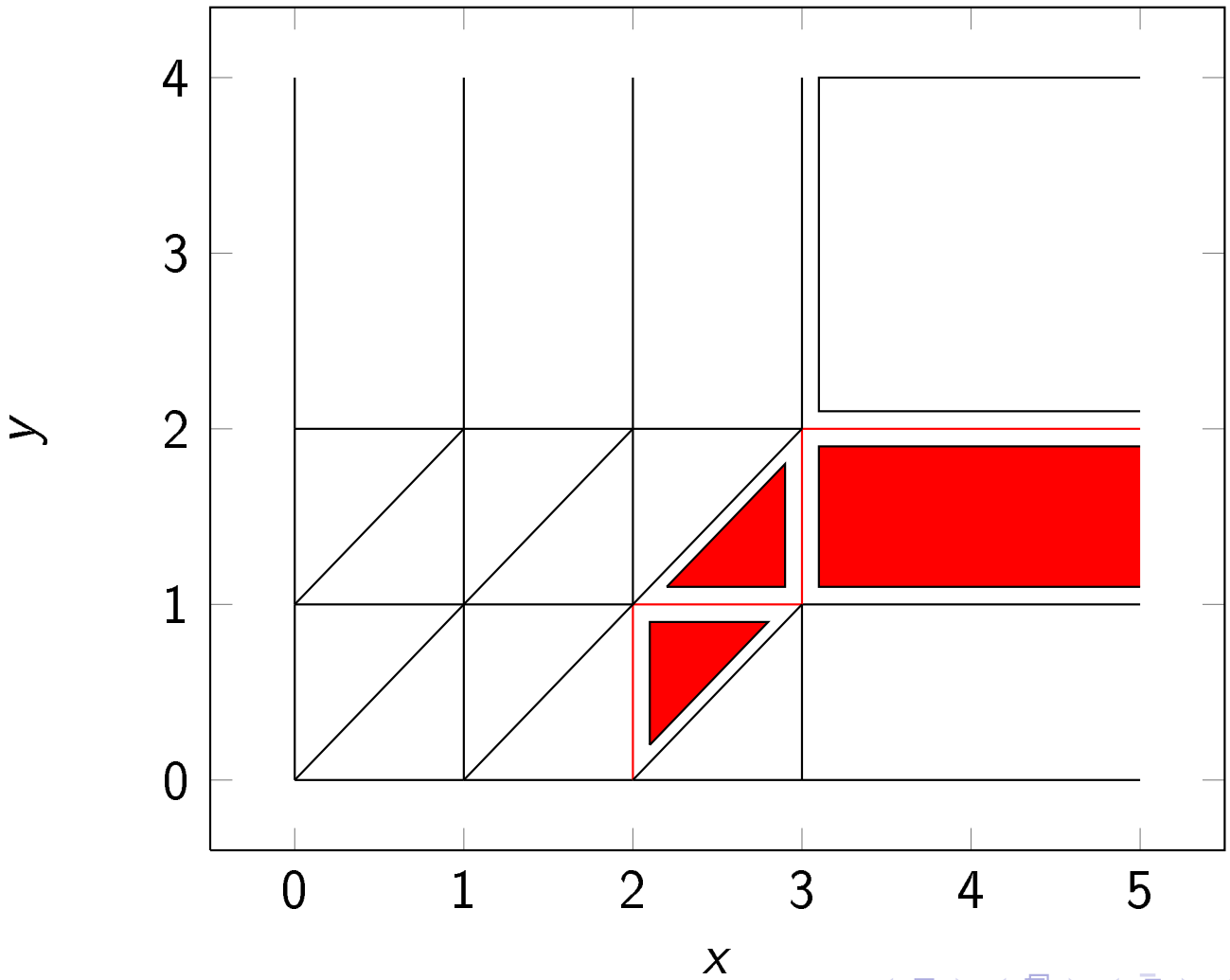
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



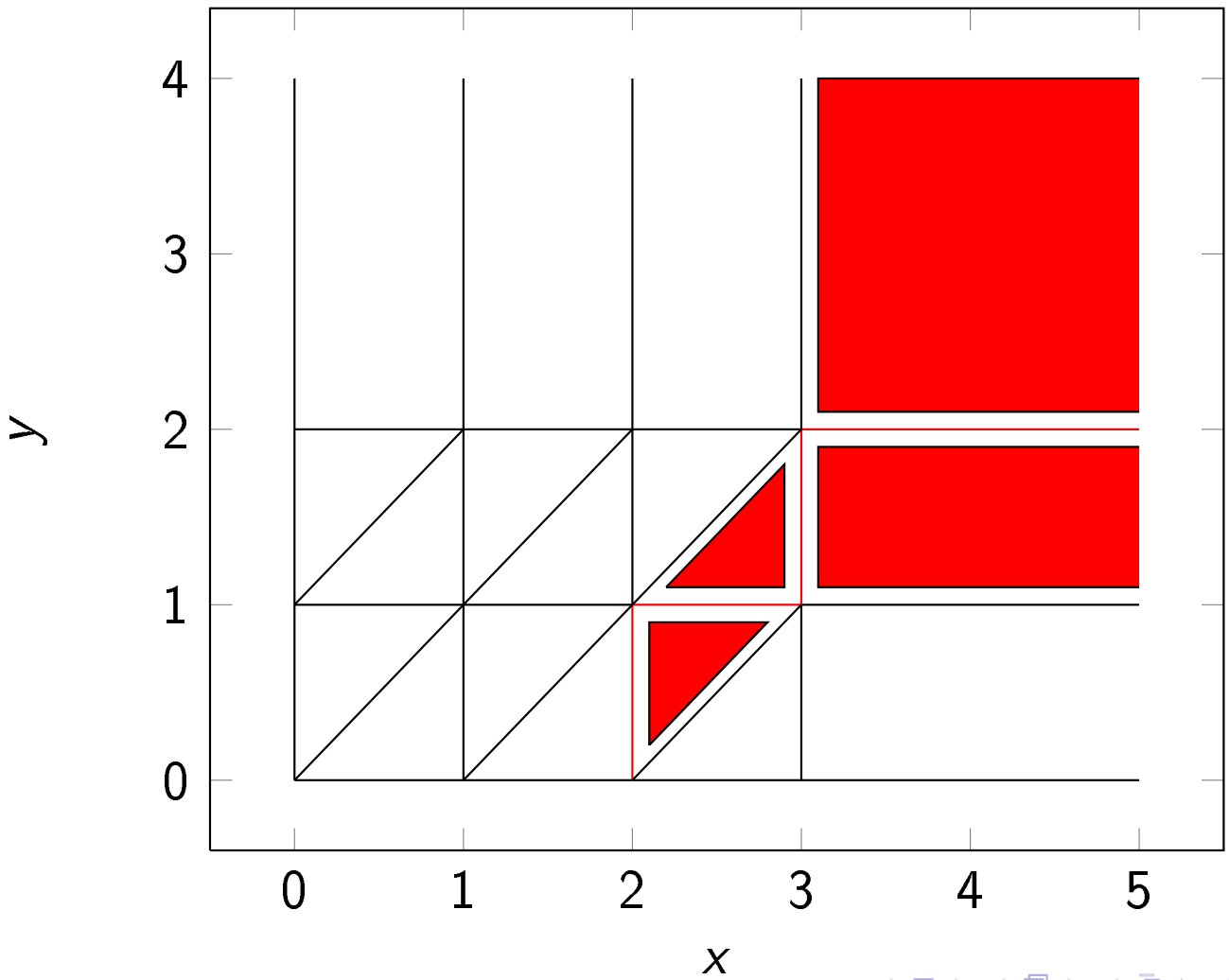
Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



Time successors in this example

What are the (time) successors of a region, i.e., how does the region change through the passing of time? We pick one region of our example:



Another example

Exercise 17

Illustrate the time successors for the starting region $\{(1, 0)\}$.

The full region graph

- Time successors are needed for simulating the delay transitions of a timed automaton.
- To define the full region graph, we also need to consider the discrete transitions. This is pretty straightforward, but we do not go into this much detail in this course.

What are regions good for?

The "equivalence" between the region graph automaton and the transitition system of a timed automaton allows us to decide basic temporal properties over timed automata.
In particular, consider two timed automaton states (ℓ, η) and (ℓ', η'), and the corresponding regions $R(\eta)$ and $R(\eta')$. Then

- (ℓ', η') is reachable from (ℓ, η) in the timed automaton if and only if
- $(\ell', R(\eta'))$ is reachable from $(\ell, R(\eta))$ in the region graph of the automaton.

Thus we have a way of deciding reachability.

Plan

1. Introductory Remarks
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Regions
   - Case Study: LEGO Mindstorm
3. Abstract Interpretation

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

One approach would be to present an example that works perfectly without any problems whatsoever . . .

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

One approach would be to present an example that works perfectly without any problems whatsoever . . .
. . . thereby hiding the fact that in the real world, things do not usually go that smoothly.

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

One approach would be to present an example that works perfectly without any problems whatsoever . . .
. . . thereby hiding the fact that in the real world, things do not usually go that smoothly.

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

One approach would be to present an example that works perfectly without any problems whatsoever . . .
. . . thereby hiding the fact that in the real world, things do not usually go that smoothly.

Instead, we will present an example that works far from perfectly . . .

What we Could Do . . .

We now want to present an example of how Uppaal can be used to verify properties of a physical system "in practice".

One approach would be to present an example that works perfectly without any problems whatsoever . . .
. . . thereby hiding the fact that in the real world, things do not usually go that smoothly.

Instead, we will present an example that works far from perfectly . . .
. . . and discuss some directions of improvement, without actually realising them.

LEGO Mindstorm©

- LEGO Mindstorm is a product by the LEGO company. It is a construction kit containing
  - a controller with a display;
  - light sensors;
  - touch sensors;
  - electric motors (rotors);
  - wheels;
  - 100s of small mechanical pieces, some resembling classical LEGO bricks.
- The box costs around 400€ and had its peek of popularity in the 2000's, but there is still a big community.
- There are dozens of programming languages for programming the controller.

A Machine for Sorting LEGO Bricks

- For our case study, we have chosen a machine for sorting LEGO bricks.
- Many such machines have been constructed. For us, it was important to choose a construction with an interesting real-time aspect.
- Our case study is inspired by [IKL+99].
- The machine has been built by Delphin Duquenne, Salim Koumad, Julien Wallart, and Antoine Willems, but the code presented here is by Jan-Georg Smaus.
- We use the NXC language.

Exercise 18

Search the web for videos of sorting machines constructed from Lego mindstorm. Post a link of a machine you find interesting, and discuss whether it has a particularly interesting real-time aspect, or not.

The Purpose of the Machine

Sorting LEGO bricks: White bricks should be kicked off the belt, black (or any other colour) bricks should remain on the belt.

The Program (1)

```
//The speeds
#define BELTSPEED 36
#define ARMSPEED -50

// NXT 2.0 Color sensor connected to port 3.
#define COLORSENSOR SENSOR_3
```

The exact values of those speed are a matter of calibration.

Exercise 19

Why is `BELTSPEED` positive and `ARMSPEED` negative?

The Program (2)

```
task main()
{
float color = 0;
SetSensorColorFull(IN_3); //set the color sensor light on
OnFwd(OUT_A,BELTSPEED);
while (true) //never ending loop
  {
  TextOut(1,LCD_LINE1,"color ");
  color = COLORSENSOR;
  NumOut(50,LCD_LINE1,color);
  if (color == 6) //6 = white
    {
    Wait(1300);
    RotateMotor(OUT_B, ARMSPEED, 360);
    }
  }
}
```

Discussion of the Program

Exercise 20

1. When/where does the program stop the moving belt?
2. What are `TextOut` and `NumOut` good for?
3. What does the line `color = ...` do?
4. What is the time unit of NXC?
5. What does the "360" stand for?

Uppaal Model

- We will now construct a Uppaal model of this system.
- One important principle is compositionality: the system is composed of several Uppaal processes.
- Of course, an Uppaal model is a-priori an abstraction of the physical reality. For this case study, we make several radical simplifications.

Uppaal Model

- We will now construct a Uppaal model of this system.
- One important principle is compositionality: the system is composed of several Uppaal processes.
- Of course, an Uppaal model is a-priori an abstraction of the physical reality. For this case study, we make several radical simplifications.

Exercise 21

(not to be answered now, but only once you have understood the model) Try to observe what these simplifications are and discuss how serious they are!

The Decomposition

What could be the Uppaal components?

- The controller: essentially executes the program.

The Decomposition

What could be the Uppaal components?

- The controller: essentially executes the program.
- A brick: as there could be several bricks, we will use templates.
  It would be "dishonest" if we modelled black and white bricks independently each in such a way that we get the results we want; the only difference between a black brick and a white brick is in the colour! Templates help us to argue this point convincingly.

The Decomposition

What could be the Uppaal components?

- The controller: essentially executes the program.
- A brick: as there could be several bricks, we will use templates. It would be "dishonest" if we modelled black and white bricks independently each in such a way that we get the results we want; the only difference between a black brick and a white brick is in the colour! Templates help us to argue this point convincingly.
- The belt: the only reason we need it in the model is that the belt process controls that two bricks cannot be on the physical belt at the same place at the same time. Each brick "controls" its own position.
- The light sensor: all it does is receive a signal from the bricks which it passes on to the controller. Not worthwhile to define a process for that! Instead, the brick communicates directly with the controller.
- The arm: receives a "start kicking" signal from the controller.

Templates and Cheating

Exercise 22

Think of some way of **cheating**, by modelling black and white bricks differently.

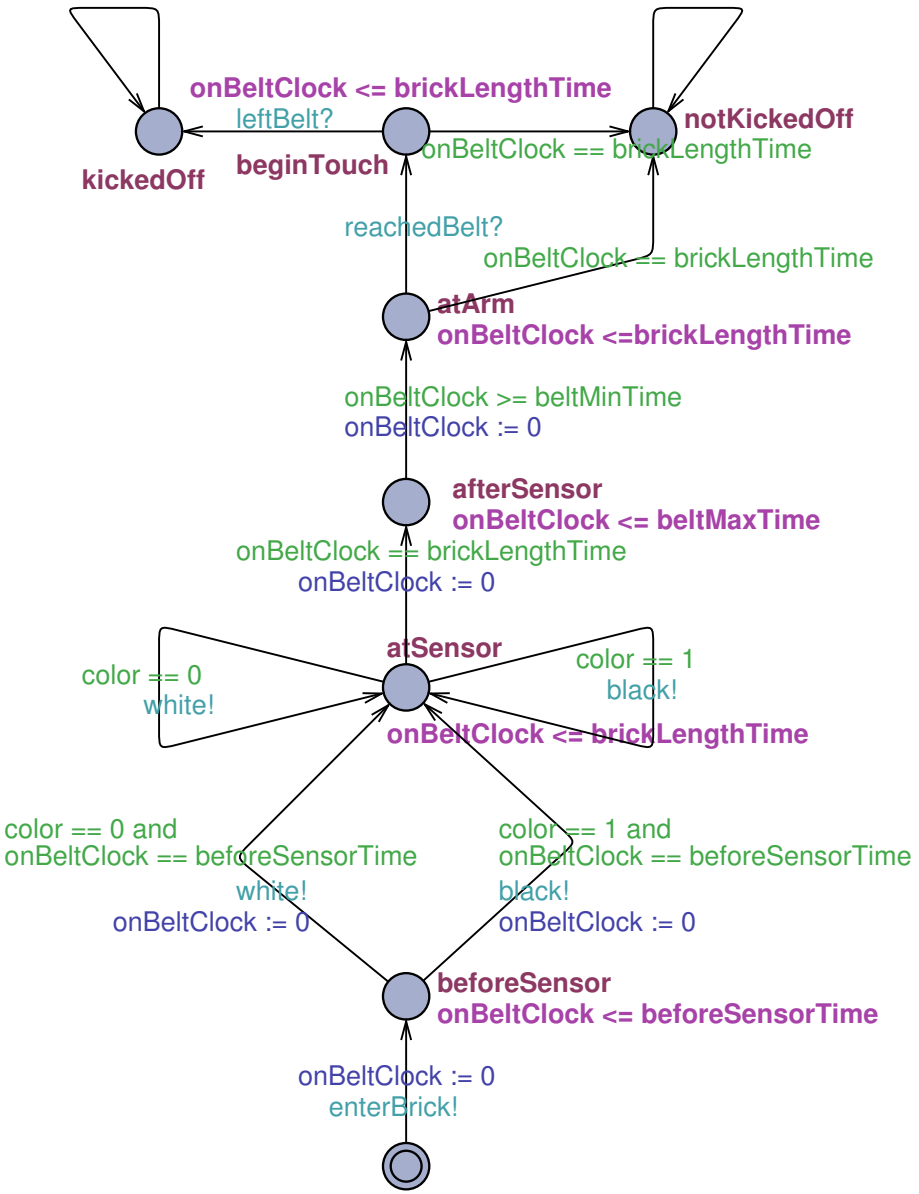
The Channels

- `white`: A white brick tells the controller: "I am beneath the sensor".
- `black`: A black brick tells the controller: "I am beneath the sensor".
- `enterBrick`: A brick tells the belt: "I start lying on you."
- `kick`: The controller tells the arm: "kick!".
- `reachedBelt`: The arm tells any brick that wants to hear it: "I reached the belt and I am ready to kick you" (broadcast channel).
- `leftBelt`: The arm tells any brick that wants to hear it: "I left the belt" (broadcast channel).

Exercise 23

Channels are for **communication** between processes. However, in some cases, the notion of "communication" is used in a strongly metaphorical sense. For which of the above channels is this particularly true?

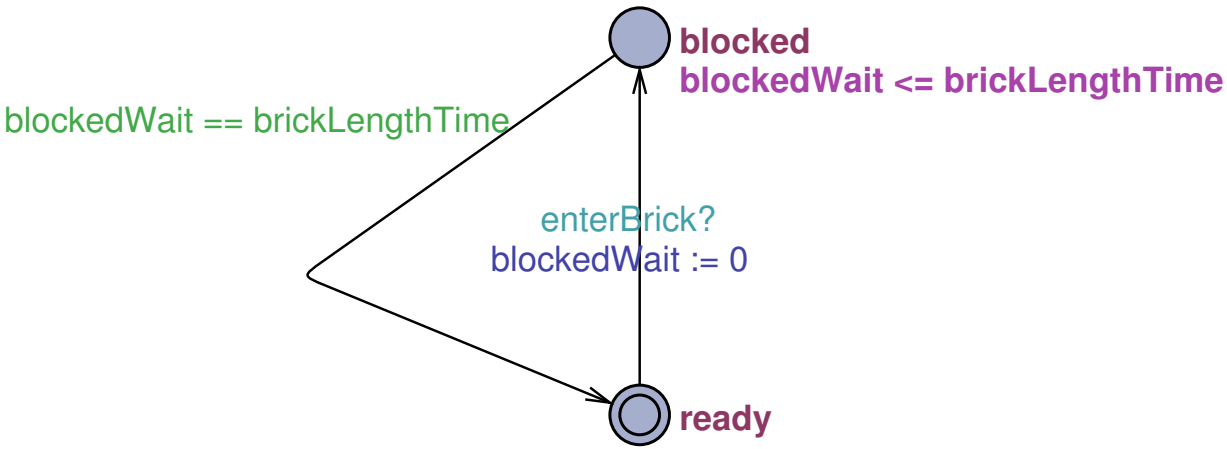
The Brick Template



Exercise 24

How does a black brick behave differently from a white brick?
In particular, does a white brick "jump off the belt" on its own?

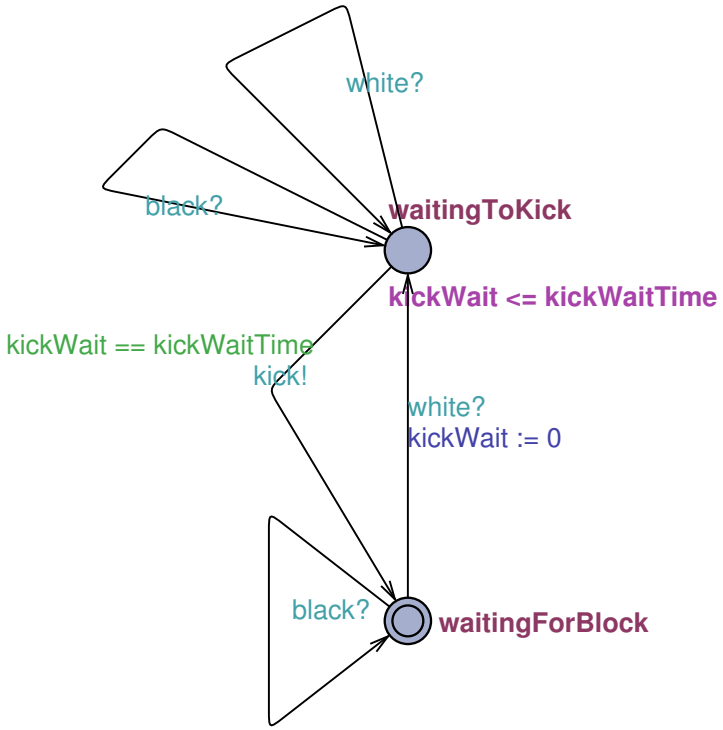
The Belt



Exercise 25

What is the belt process good for?

The Controller

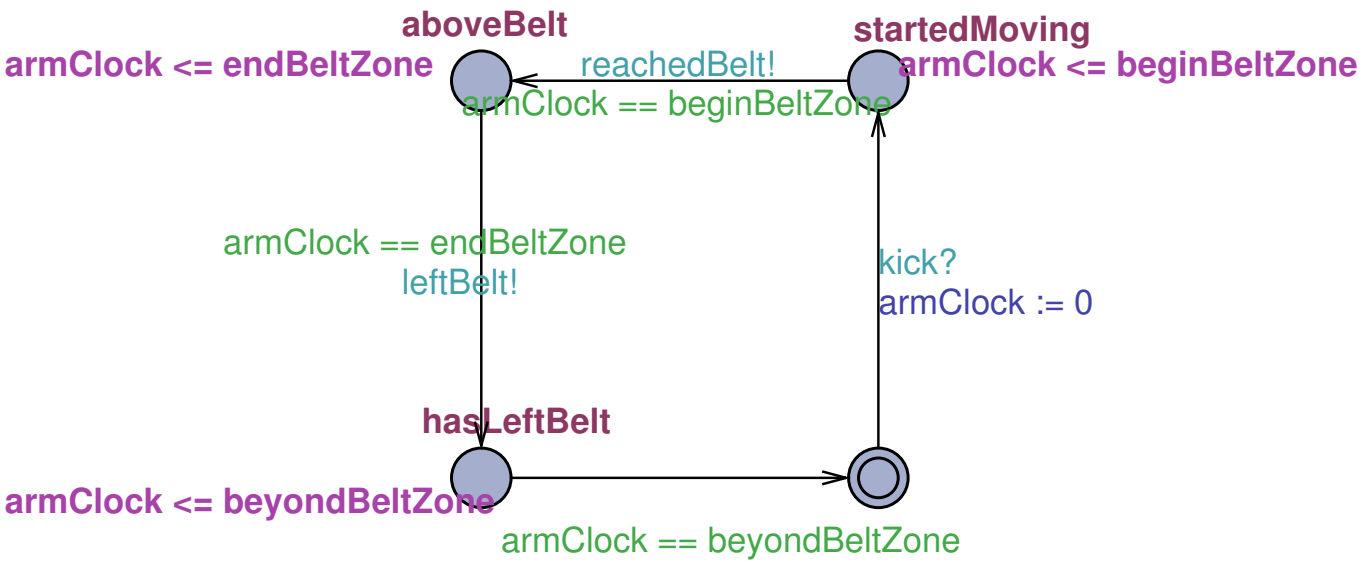


It could be envisaged that the translation of the program into the Uppaal process is done automatically as in [IKL+99].

Exercise 26

So how was the present controller model generated?

The Arm



Exercise 27

Sketch the movement of the arm and indicate the corresponding locations of the above process.

Sanity Check for Synchronisation

Wherever we have a process that might want to send a signal and cannot do so because there is no recipient available, we must ask: does this blocking correspond to reality?
For example:

- The arm has started moving and is just about to reach the zone above the belt. Should it be blocked because there is no brick ready to receive its `reachedBelt` signal?

Sanity Check for Synchronisation

Wherever we have a process that might want to send a signal and cannot do so because there is no recipient available, we must ask: does this blocking correspond to reality?
For example:

- The arm has started moving and is just about to reach the zone above the belt. Should it be blocked because there is no brick ready to receive its `reachedBelt` signal?
- A brick wants to enter the belt. Should it be blocked because there is currently still another brick at the beginning of the belt?

Sanity Check for Synchronisation

Wherever we have a process that might want to send a signal and cannot do so because there is no recipient available, we must ask: does this blocking correspond to reality?
For example:

- The arm has started moving and is just about to reach the zone above the belt. Should it be blocked because there is no brick ready to receive its `reachedBelt` signal?
- A brick wants to enter the belt. Should it be blocked because there is currently still another brick at the beginning of the belt?

Exercise 28

Answer the above questions.

Setting the Parameters

Setting the parameters must be done by down-to-earth chronometric and geometric measurements (see TP!).

What Properties?

What properties might one want to prove?

- Deadlock freedom
- It is possible for a white brick to be kicked off and it is impossible for it to reach the end of the belt.
- It is possible for a black brick to reach the end of the belt and it is impossible for it be kicked off.
- If a white brick enters the belt, it will eventually be kicked off.
- If a black brick enters the belt, it will eventually reach the end of the belt.

All of the above for the following scenarii, if applicable:

- There is exactly one white brick in the game.
- There is exactly one black brick in the game.
- There are exactly one black and one white brick in the game.
- There are two white bricks and one black brick in the game.

See TP!

Lessons

- Simple behaviours (e.g., just one brick!) can be found by observation or by trusting the semantics of NXC (e.g., `Wait(1300)` will cause the controller to wait exactly 130ms). This can be used to design each process.
- Uppaal can detect that by the complex interaction of those simple behaviours, phenomena may occur (i.e., states are reachable) that one might not discover by physical experiments.
- The crucial question is: have we really modelled the interfaces between the process faithfully enough?
  If not, it might turn out that while our abstractions are good enough for the single processes, they are not good enough for the composition (e.g., one bricks vs. several bricks).
  If yes, we obtain a guarantee we can trust.

Improvements

On the model side:

- Relative correctness: if the light sensor captures the signal, the white brick will be kicked off.
- Probabilistic verification
- Be more faithful: include tolerances in many places.

On the physical side:

- Improve the arm.

Conclusion

- Timed automata are a modelling framework for systems where real time matters, i.e., where we are interested in events that happen at a particular time.
- We have only looked at few examples and no realistic ones, but there exist countless examples.
- The definition of timed automata is quite restrictive, e.g., it is not possible to have stopwatches, or to have clocks that run at a different speed ...
- Thanks to these restrictions, timed automata are accessible to automatic verification.
- The tool Uppaal proves it.

Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

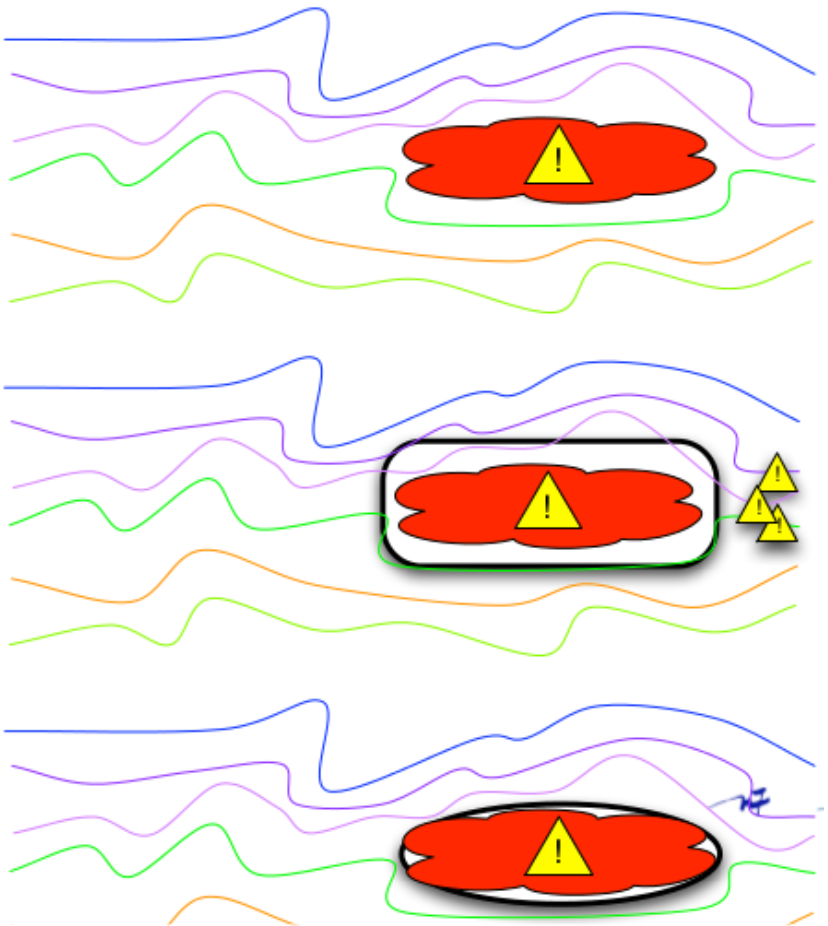
Acknowledgements

The slides and exercises of this chapter are based on material by Loïc Correnson, Nikolai Kosmatov, A. Miné, and Pierre Roux, and on [KKP+15].

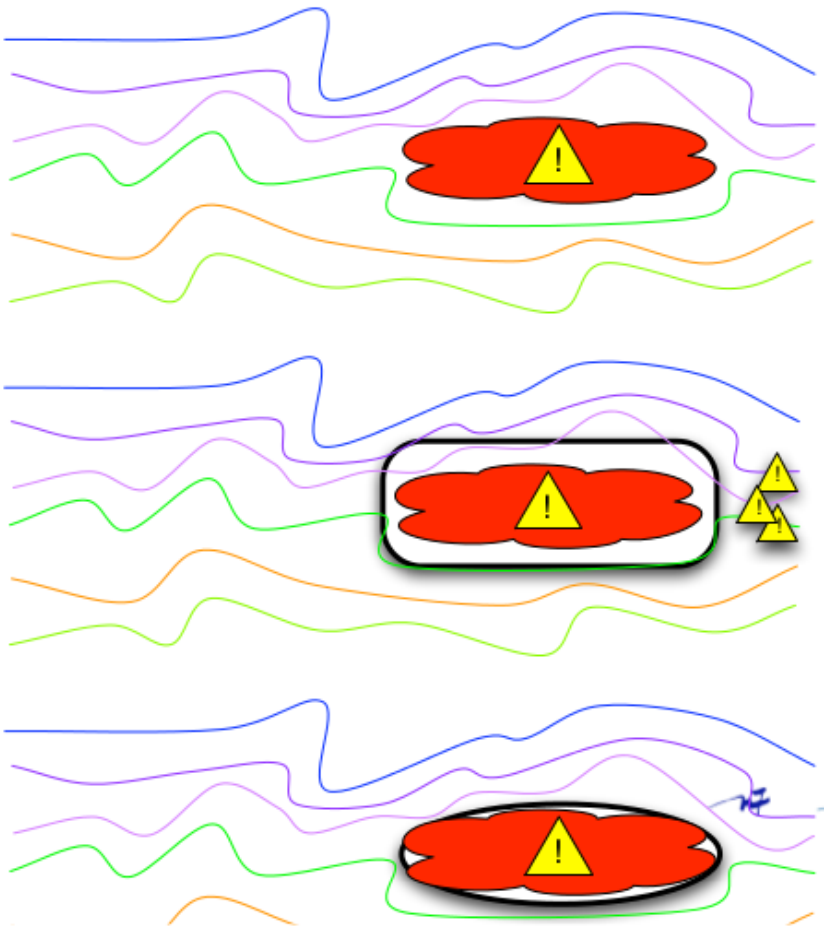
Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

Abstract Interpretation at a glance

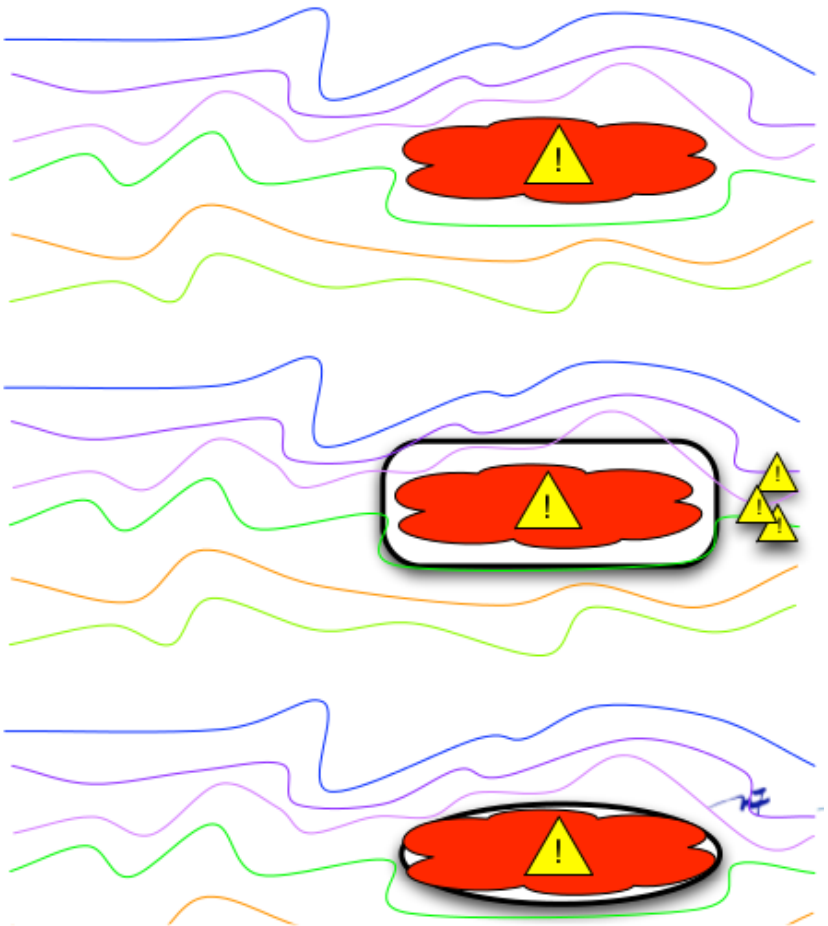


Abstract Interpretation at a glance



$$_0x = \mathbf{rand}(0, 12)\,;_1 y = 42\,;$$
$$\mathbf{while}\ _2(x > 0)\ \{$$
$$\quad _3x = x - 2\,;$$
$$\quad _4y = y + 4\,;$$
$$\}_5$$

Abstract Interpretation at a glance



$$_0x = \textbf{rand}(0, 12)\,;_1 y = 42\,;$$
$$\textbf{while}\ _2(x > 0)\ \{$$
$$\quad _3x = x - 2\,;$$
$$\quad _4y = y + 4\,;$$
$$\}_5$$

Goal: Infer information for possible values of all variables at each program point, e.g. that at point 2, $x \in [-1, 12]$.

Image on the left

On top, the coloured lines depict the executions of a system, and the red cloud is a dangerous zone. The system is good.
In the middle, we work with an approximation of the dangerous zone that wrongly suggests that the system is not good.
At the bottom, we work with a refined approximation of the dangerous zone that shows that the system is good.

Exercise 29

Why might we want to work with an approximation of the dangerous zone, bigger than the actual dangerous zone? What is the "advantage" of the square (with rounded corners) of the middle image, compared to the cloud?

Basic Idea

- Abstract interpretation was introduced by [CC77] and is a form of static analysis.
- Idea: replace computation on concrete data by computation on abstract data so that the abstract computation overaproximates "cheaply" the concrete computation.

Running example: `P` (positive integers), `N` (negative ...), `Z` (0), `PZ` (non-negative), `NZ` (non-positive), `PNZ` (all).

An example: Euclid's algorithm

```
int x = a, y = b;
int s = 1, t = 0, u = 0, v = 1;
  while(y > 0){
    int r = x % y;
    int q = x / y;
    x = y;
    y = r;
    int w = u;
    u = s - u * q;
    s = w;
    w = v;
    v = t - v * q;
    t = w;
    }
return x;
```

Exercise 30

What does this program compute?
What are s, t, u, v?

Abstraction of example

```
int x = P, y = P;
int s = P, t = Z, u = Z, v = P;
  while(y > Z){
    int r = x % y;
    int q = x / y;
    x = y;//x = P is invariant
    y = r;//y = PZ is invariant
    int w = u;
    u = s - u * q;//alternates between PZ and NZ
    s = w;//alternates between PZ and NZ
    w = v;
    v = t - v * q;//alternates between PZ and NZ
    t = w;//alternates between PZ and NZ
    }
return x;
```

Let's be concrete!

- We consider a domain $\mathcal{D}$ of concrete objects.
- For instance, consider $\mathcal{D} = 2^{\mathbb{Z}}$. Intuition: the set of values a given variable can take at a given program point.

Let's be concrete!

- We consider a domain $\mathcal{D}$ of concrete objects.
- For instance, consider $\mathcal{D} = 2^{\mathbb{Z}}$. Intuition: the set of values a given variable can take at a given program point.
- This is just one example, and even in this course, we will discuss a more general instance of concrete domain.

Exercise 31

What is $2^{\mathbb{Z}}$?

Let's be concrete!

- We consider a domain $\mathcal{D}$ of concrete objects.
- For instance, consider $\mathcal{D} = 2^{\mathbb{Z}}$. Intuition: the set of values a given variable can take at a given program point.
- This is just one example, and even in this course, we will discuss a more general instance of concrete domain.

Exercise 31

What is $2^{\mathbb{Z}}$?

Operations on concrete domain defined in "natural" way, where errors lead to absence of results, e.g.:

- $\{1,4\} + \{5,9\} = \{6,10,9,13\}$;
- $\{10,20\}/\{-2,0,2\} = \{-10,-5,5,10\}$
- $\{10,20\}/\{0\} = \emptyset$

Exercise 32

$\{1,4\} \times \{5,9\} = \ldots$?

Order $\sqsubseteq$ between concrete objects

Definition (order $\sqsubseteq$ between concrete objects)

For the concrete domain above, an object o' overapproximates an object o if $o \subseteq o'$ (i.e., $\sqsubseteq$ is defined as $\subseteq$).

Order $\sqsubseteq$ between concrete objects

Definition (order $\sqsubseteq$ between concrete objects)

For the concrete domain above, an object o' overapproximates an object o if $o \subseteq o'$ (i.e., $\sqsubseteq$ is defined as $\subseteq$).

Example: $\{1,2\} \sqsubseteq \{1,2,5\}$. "$\{1,2,5\}$ overapproximates $\{1,2\}$."

Aside: Intervals

As an ad-hoc shorthand for writing certain sets of integers, we can use intervals, e.g. $\{3, 4, 5\} = [3, 5]$.

Aside: Intervals

As an ad-hoc shorthand for writing certain sets of integers, we can use intervals, e.g. $\{3,4,5\} = [3,5]$.
We also sometimes use the notation

$$n\mathbb{Z} + m := \{n \cdot x + m \mid x \in \mathbb{Z}\}$$

Exercise 33

1. $[1,5] = \ldots?$
2. $\{2,3,4,5,7\} \sqsubseteq [2,7]?$
3. $3\mathbb{Z} + 5 \sqsubseteq 6\mathbb{Z} + 5?$

Abstract Domains

Definition (Abstract domain $\mathcal{D}^\sharp$)

An abstract domain specifies:

- a set $\mathcal{D}^\sharp$ of abstract objects;
- abstract operations that mimic in the abstract the concrete operations on $\mathcal{D}$.

Abstract Domains

Definition (Abstract domain $\mathcal{D}^\sharp$)

An abstract domain specifies:

- a set $\mathcal{D}^\sharp$ of abstract objects;
- abstract operations that mimic in the abstract the concrete operations on $\mathcal{D}$.

Example: $\mathcal{D}^\sharp = \{\mathtt{P}, \mathtt{N}, \mathtt{Z}, \mathtt{PZ}, \mathtt{NZ}, \mathtt{PNZ}\}$.
$\mathtt{P} +^\# \mathtt{P} = \mathtt{P}$ ("positive + positive = positive"), ...

Exercise 34

$\mathtt{N} \times^\# \mathtt{N} = \ldots ?$ $\mathtt{N} \times^\# \mathtt{P} = \ldots ?$ $\mathtt{N} \times^\# \mathtt{Z} = \ldots ?$

Abstractions

Definition (abstraction α)

An abstraction (function) α maps each concrete object o to an abstract object $o^\sharp$, which is a simplification of o.

Abstractions

Definition (abstraction α)

An abstraction (function) α maps each concrete object o to an abstract object $o^\sharp$, which is a simplification of o.

Example: $\alpha(\{1,7\}) = \texttt{P}$, $\alpha(\{1,7,9,10\}) = \texttt{P}$, ...

Exercise 35

$\alpha(\{0,1,7\}) = \ldots?$ $\alpha(\{-1,-7\}) =?$ $\alpha(\{-7,1\}) = \ldots?$

Concretisations

Definition (concretisation γ)

A concretisation (function) γ maps each abstract object $o^\sharp$ to the greatest (wrt. $\sqsubseteq$) concrete object o such that $\alpha(o) = o^\sharp$.

Concretisations

Definition (concretisation γ)

A concretisation (function) γ maps each abstract object $o^\sharp$ to the greatest (wrt. $\sqsubseteq$) concrete object o such that $\alpha(o) = o^\sharp$.

Example: What is $\gamma(\mathtt{P})$?

Concretisations

Definition (concretisation γ)

A concretisation (function) γ maps each abstract object $o^\sharp$ to the greatest (wrt. $\sqsubseteq$) concrete object o such that $\alpha(o) = o^\sharp$.

Example: What is $\gamma(\mathtt{P})$?
We have $\alpha(\{1,7\}) = \mathtt{P}$, $\alpha(\{1,7,9,10\}) = \mathtt{P}$, $\alpha(\{1,7,9,10,11,25\}) = \mathtt{P}$, ..., but the greatest set $\in 2^{\mathbb{Z}}$ (the greatest concrete object) o such that $\alpha(o) = \mathtt{P}$ is the set $\{1,2,3,\ldots\}$. Hence $\gamma(\mathtt{P}) = \{1,2,3,\ldots\}$.

Exercise 36

$\gamma(\mathtt{N}) = \ldots?$ $\gamma(\mathtt{Z}) = \ldots?$

Order $\sqsubseteq^\sharp$ between abstract objects

We define an abstract order $\sqsubseteq^\sharp$ as follows:

Definition ($\sqsubseteq^\sharp$)

$$\forall o^\sharp, o^{\sharp\prime}, \quad o^\sharp \sqsubseteq^\sharp o^{\sharp\prime} :\Leftrightarrow \gamma(o^\sharp) \sqsubseteq \gamma(o^{\sharp\prime})$$

Order $\sqsubseteq^\sharp$ between abstract objects

We define an abstract order $\sqsubseteq^\sharp$ as follows:

Definition ($\sqsubseteq^\sharp$)

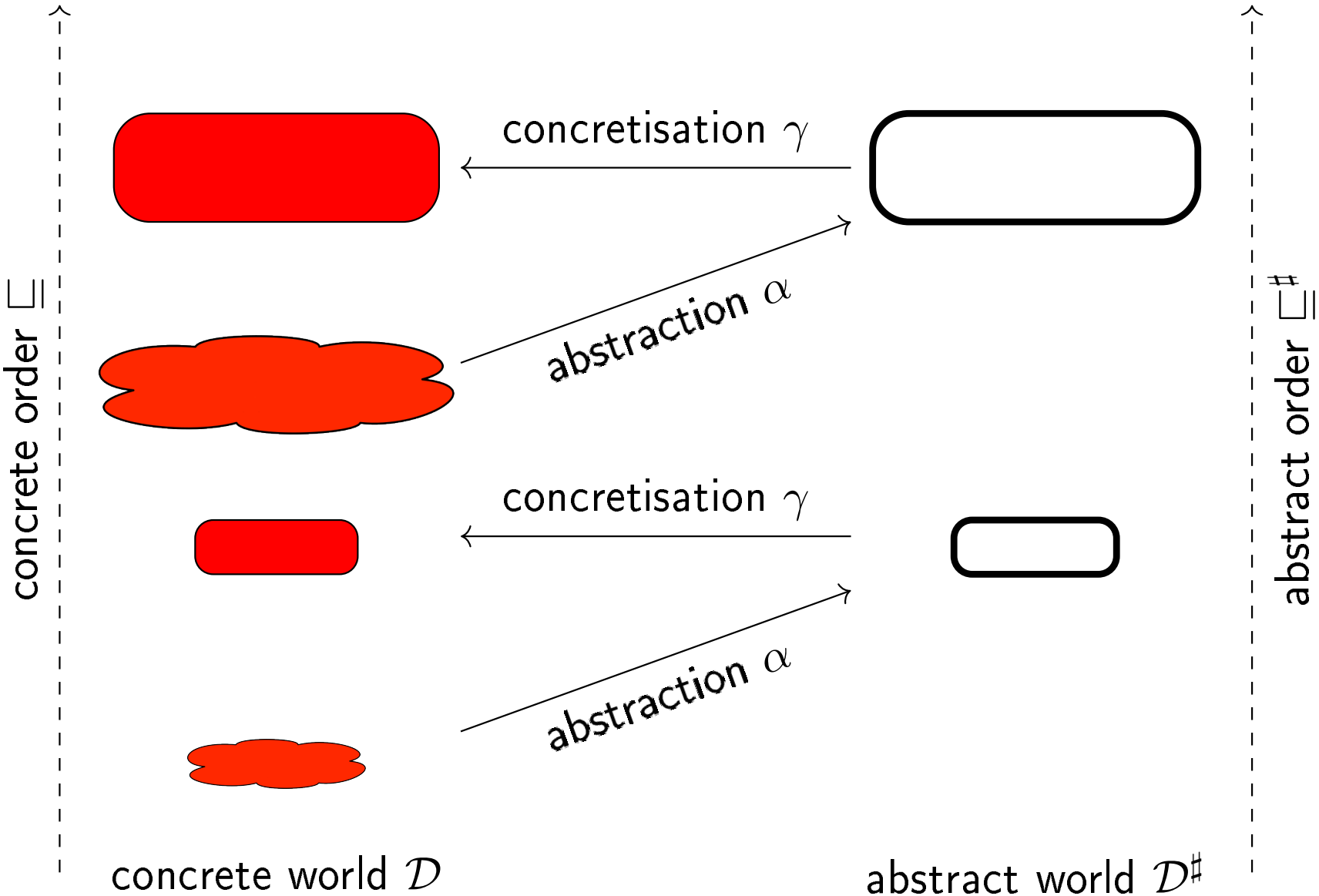
$$\forall o^\sharp, o^{\sharp\prime}, \quad o^\sharp \sqsubseteq^\sharp o^{\sharp\prime} :\Leftrightarrow \gamma(o^\sharp) \sqsubseteq \gamma(o^{\sharp\prime})$$

Example: $\mathtt{P} \sqsubseteq^\sharp \mathtt{PZ}$ because $\gamma(\mathtt{P}) = \{1, 2, 3, \ldots\} \sqsubseteq \{0, 1, 2, 3, \ldots\} = \gamma(\mathtt{PZ})$.

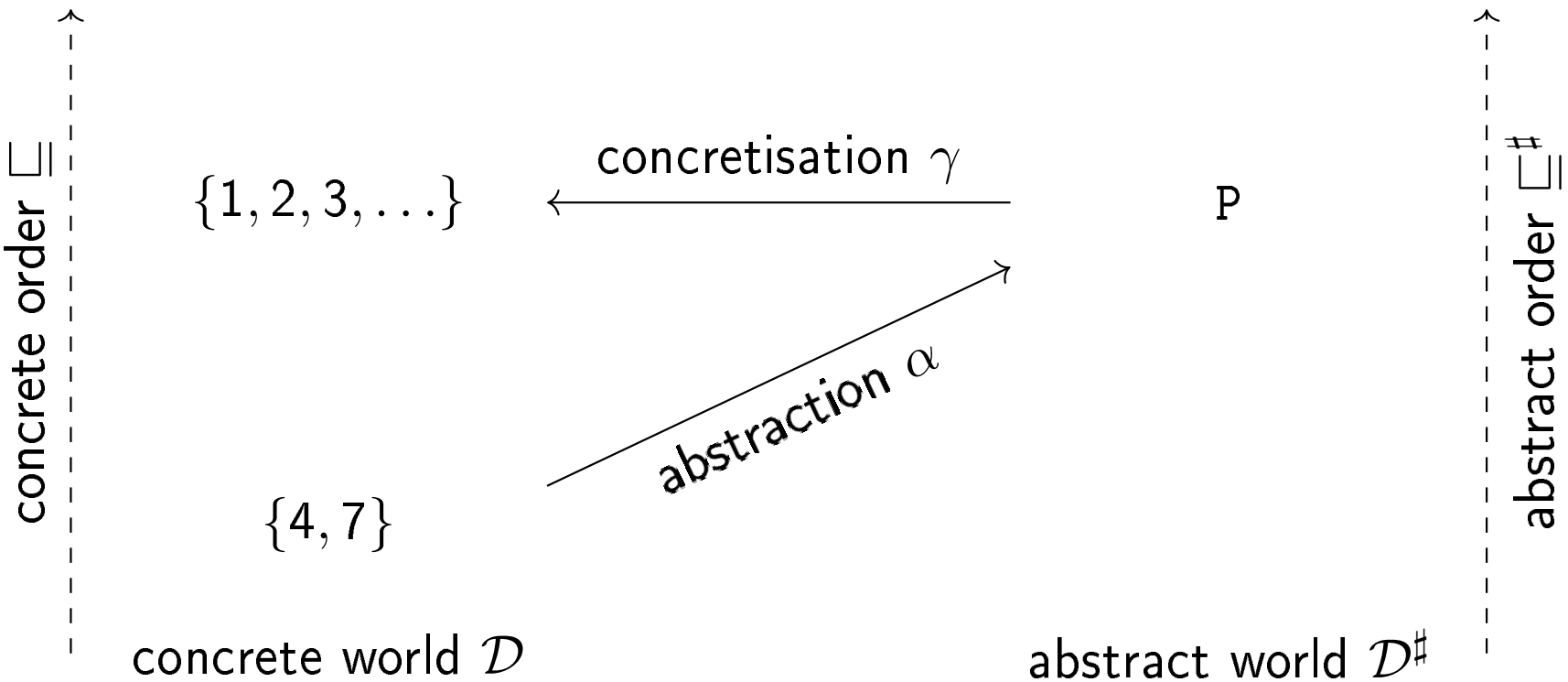
Exercise 37

Draw the entire $\sqsubseteq^\sharp$-lattice for $\{\mathtt{P}, \mathtt{Z}, \mathtt{N}, \mathtt{PZ}, \mathtt{NZ}, \mathtt{PNZ}\}$.

Concrete — abstract : summary



Concrete — abstract : summary on PNZ-example



Abstract operations

To manipulate abstract objects we need to define abstract operations that correctly mimic the concrete operations.
$unary^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ or $binary^\sharp : (\mathcal{D}^\sharp \times \mathcal{D}^\sharp) \to \mathcal{D}^\sharp$
This is done as follows.

- $unary^\sharp(x) = \alpha(unary(\gamma(x)))$
- $binary^\sharp(x, y) = \alpha(binary(\gamma(x), \gamma(y)))$
- ...

We will get back to this later ...

Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

A toy language

Syntax

$$
\begin{aligned}
\mathit{stm} ::= {} & v = \mathit{expr}; \mid \mathit{stm}\ \mathit{stm} \\
& \mid \textbf{if}\ (\mathit{expr} > 0)\ \{\ \mathit{stm}\ \}\ \textbf{else}\ \{\ \mathit{stm}\ \} \\
& \mid \textbf{while}\ (\mathit{expr} > 0)\ \{\ \mathit{stm}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{expr} ::= {} & v \mid n \mid \textbf{rand}(n, n) \\
& \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} - \mathit{expr} \mid \mathit{expr} \times \mathit{expr} \mid \mathit{expr} / \mathit{expr}
\end{aligned}
$$

$v \in \mathbb{V}$, a set of variables

$n \in \mathbb{Z}$

rand(n_1, n_2) simulates an input value.

Exercise 38

Write a naive program for computing $\lfloor \sqrt{n} \rfloor$ (0 if $n < 0$).

Toy language example

Example

$x = \textbf{rand}(0, 12); y = 42;$

while $(x > 0)$ **{**

$\quad x = x - 2;$

$\quad y = y + 4;$

}

A run
(values at loop entry point) :

x	7	5	3	1	-1
y	42	46	50	54	58

Remarks

- very simple language, without functions,
- but representative of an imperative language like C
- it's Turing-complete

Towards program semantics

So far we have seen how concrete data (numbers, sets of numbers) are abstracted, and how operations on abstract data objects mimic operations on concrete data objects.

Towards program semantics

So far we have seen how concrete data (numbers, sets of numbers) are abstracted, and how operations on abstract data objects mimic operations on concrete data objects.
This will be the basis for now looking at program semantics and abstractions. Important aspects:

- Variables and their values (memory states);
- program points (we denote the set of program points by L).

Concrete semantics, expressions

Expressions are evaluated w.r.t. a memory state (environment), i.e., a function that assigns a value (for simplicity: $\in \mathbb{Z}$) to each variable, i.e., a function that lives in $\mathbb{V} \to \mathbb{Z}$.

Concrete semantics, expressions

Expressions are evaluated w.r.t. a memory state (environment), i.e., a function that assigns a value (for simplicity: $\in \mathbb{Z}$) to each variable, i.e., a function that lives in $\mathbb{V} \rightarrow \mathbb{Z}$.

One might expect that the value of an expression is simply a number in $\mathbb{Z}$, but due to the presence of a random number generator, it is actually a set of numbers in $\mathbb{Z}$.

Concrete semantics, expressions

Expressions are evaluated w.r.t. a memory state (environment), i.e., a function that assigns a value (for simplicity: $\in \mathbb{Z}$) to each variable, i.e., a function that lives in $\mathbb{V} \to \mathbb{Z}$.
One might expect that the value of an expression is simply a number in $\mathbb{Z}$, but due to the presence of a random number generator, it is actually a set of numbers in $\mathbb{Z}$. So here is the signature of the semantics of expressions : $[\![e]\!]_{\mathrm{E}} : (\mathbb{V} \to \mathbb{Z}) \to 2^{\mathbb{Z}}$

$$
\begin{aligned}
[\![v]\!]_{\mathrm{E}}(\rho) &= \{\rho(v)\} \\
[\![n]\!]_{\mathrm{E}}(\rho) &= \{n\} \\
[\![\mathbf{rand}(n_1, n_2)]\!]_{\mathrm{E}}(\rho) &= \{n \in \mathbb{Z} \mid n_1 \leq n \leq n_2\} \\
[\![e_1 + e_2]\!]_{\mathrm{E}}(\rho) &= \{n_1 + n_2 \mid n_1 \in [\![e_1]\!]_{\mathrm{E}}(\rho) \wedge n_2 \in [\![e_2]\!]_{\mathrm{E}}(\rho)\} \\
&\ldots
\end{aligned}
$$

Exercise 39

Let $\rho = \{x \mapsto 2, y \mapsto 6\}$.
$[\![x + y]\!]_{\mathrm{E}}(\rho) = \ldots ?$

Type of the concrete program semantics

The concrete semantics is of this type:

$$L \to 2^{\mathbb{V} \to \mathbb{Z}}$$

Type of the concrete program semantics

The concrete semantics is of this type:

$$L \to 2^{\mathbb{V} \to \mathbb{Z}}$$

- A function that to each program point (in L)
- maps a set of possible memory states:
  - a function that to each variable (in $\mathbb{V}$)
  - maps its value in memory (in $\mathbb{Z}$)

Example

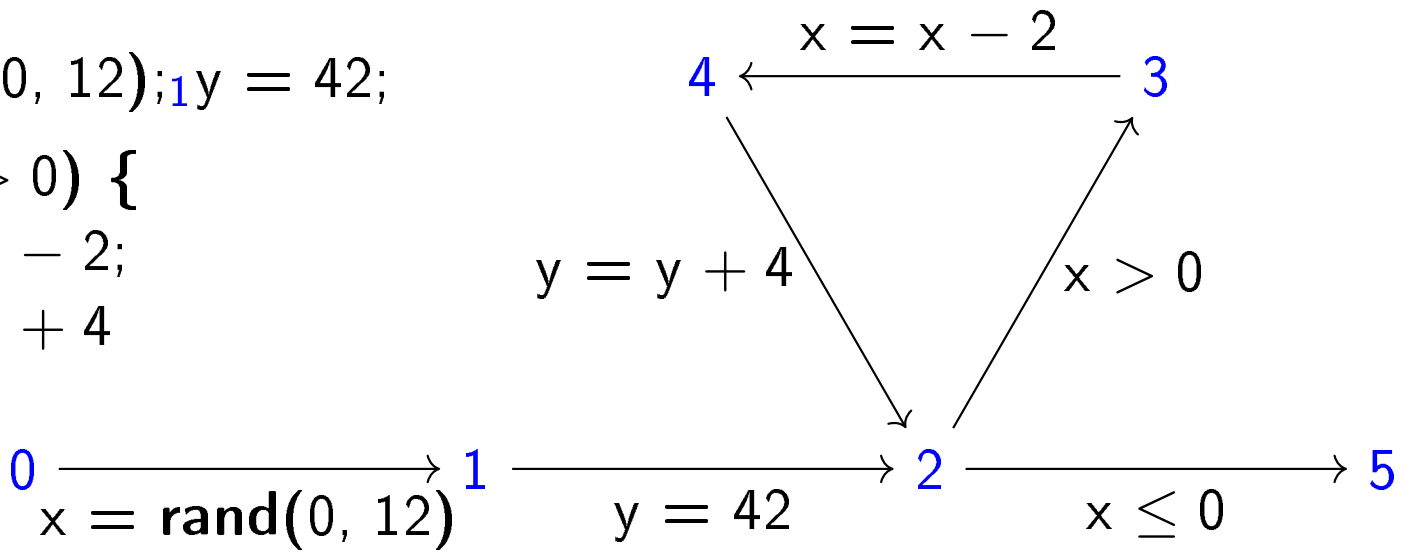
$_0x = \mathbf{rand}(0, 12);{}_1y = 42;$

$\mathbf{while}\ {}_2(x > 0)\ \{$

$\quad {}_3x = x - 2;$

$\quad {}_4y = y + 4$

$\}_5$



Denoting by S_i the semantics at point i:

$$S_0 = \mathbb{V} \to \mathbb{Z} \qquad (\mathbb{V} = \{x, y\})$$
$$S_1 = \{f \in (\mathbb{V} \to \mathbb{Z}) \mid f(x) \in [0, 12]\}$$
$$S_2 = \{f \mid f(x) \in [-1, 12], f(y) \in \{42, 46, \ldots, 62, 66\}, 2f(x) + f(y) \in [42, 66]\}$$
$$S_3 = \{f \mid f(x) \in [1, 12], f(y) \in \{42, 46, \ldots, 62, 66\}, 2f(x) + f(y) \in [42, 66]\}$$
$$S_4 = \{f \mid f(x) \in [-1, 10], f(y) \in \{42, 46, \ldots, 62, 66\}, 2f(x) + f(y) \in [38, 62]\}$$
$$S_5 = \{f \mid f(x) \in [-1, 0], f(y) \in \{42, 46, \ldots, 62, 66\}, 2f(x) + f(y) \in [42, 66]\}$$

Exercise 40

Is $(x = 10, y = 46)$ possible at point 2? Is $(x = 10, y = 54)$ possible?

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point

  $\Rightarrow$ we keep it.

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point

  $\Rightarrow$ we keep it.

- $\mathbb{V}$ is finite and we are interested in all the variables

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point
  $\Rightarrow$ we keep it.
- $\mathbb{V}$ is finite and we are interested in all the variables
  $\Rightarrow$ we keep them.

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point
  $\Rightarrow$ we keep it.
- $\mathbb{V}$ is finite and we are interested in all the variables
  $\Rightarrow$ we keep them.
- $\mathbb{Z}$ (and hence the set of functions $\mathbb{V} \to \mathbb{Z}$) is infinite

[1] But even abstract domains can be infinite and require further abstraction techniques.

What exactly are we abstracting?

The concrete semantics is uncomputable, we want to simplify it. But what exactly are we simplifying?

- L is finite and we would like to know what happens at each program point
  $\Rightarrow$ we keep it.
- $\mathbb{V}$ is finite and we are interested in all the variables
  $\Rightarrow$ we keep them.
- $\mathbb{Z}$ (and hence the set of functions $\mathbb{V} \to \mathbb{Z}$) is infinite
  $\Rightarrow$ this is what we are abstracting.[1]

Exercise 41

Is $x = 10$ possible at point 2? Is $y = 54$ possible at point 2?

[1] But even abstract domains can be infinite and require further abstraction techniques.

How to abstract $2^{\mathbb{V}\to\mathbb{Z}}$?

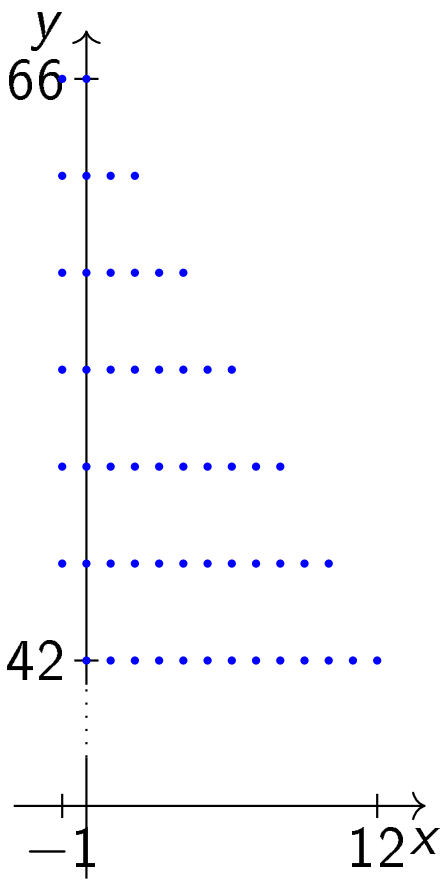
Abstract $2^{\mathbb{V}\to\mathbb{Z}}$ into $\mathbb{V}\to 2^{\mathbb{Z}}$, and then $2^{\mathbb{Z}}$ into one $\mathcal{D}^\sharp$. Note that the values of x and y are independent, i.e., any actual dependencies are lost!

Exercise 42

Discuss the previous phrase using the two exercises above.

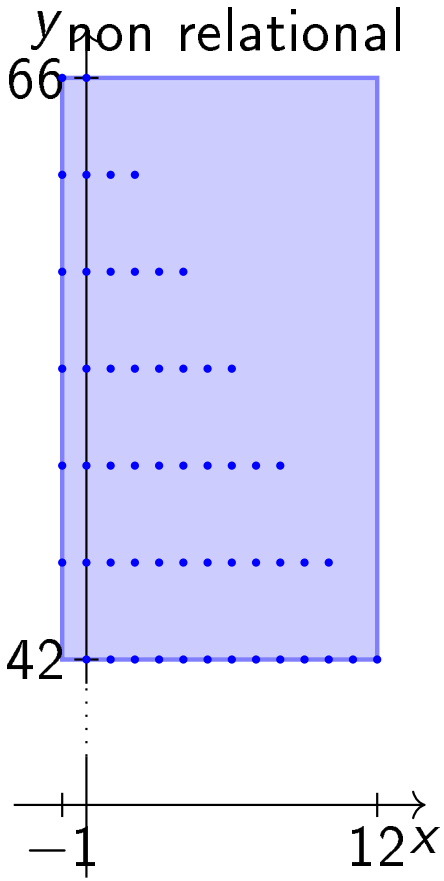
Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



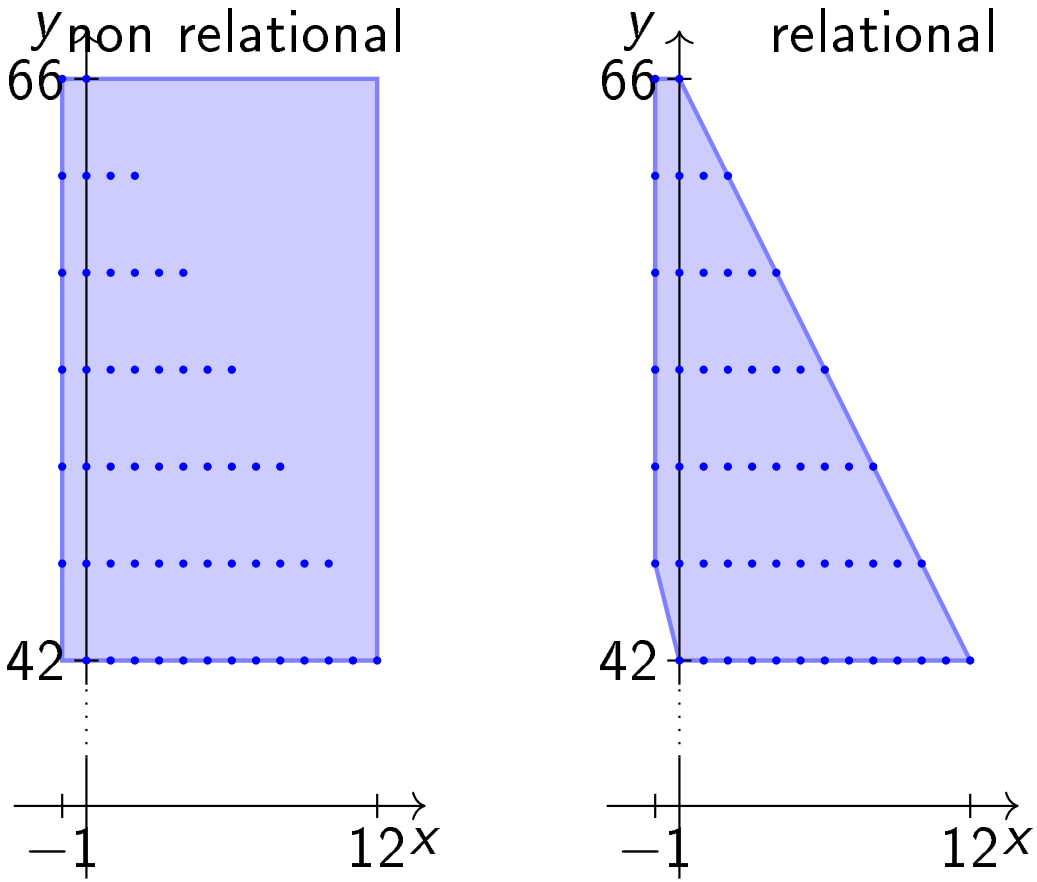
Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



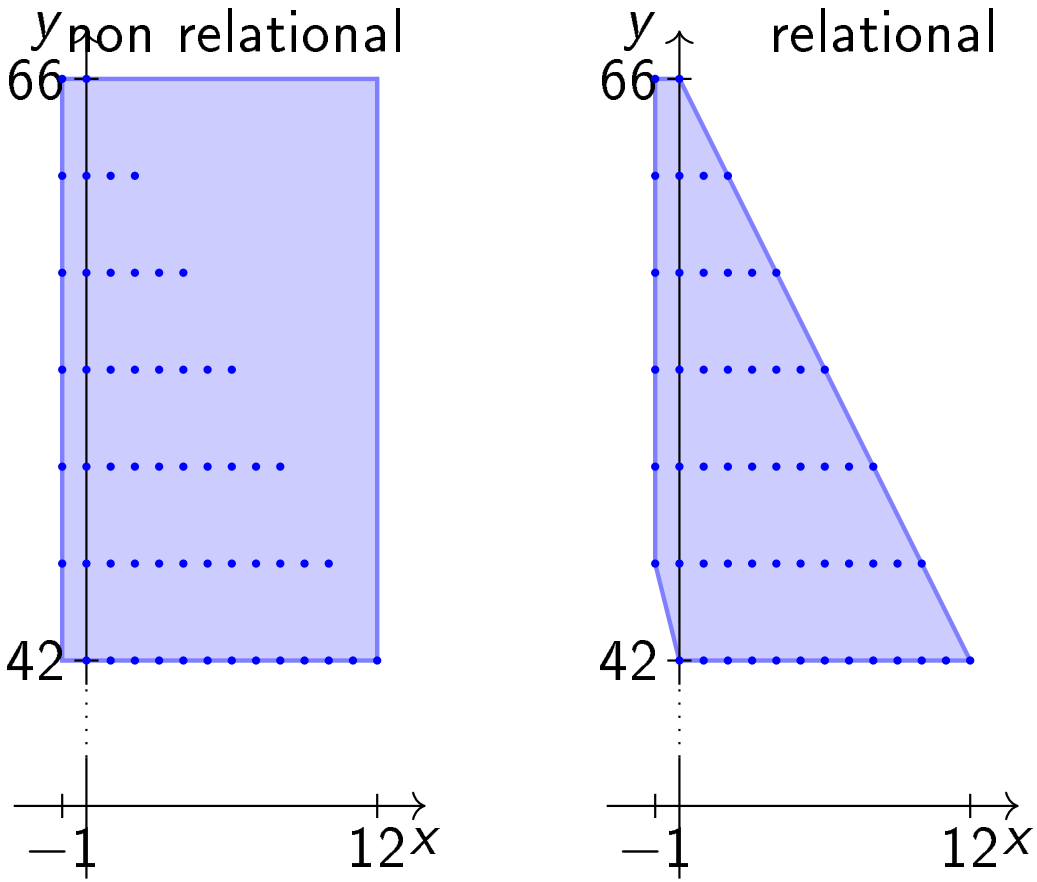
Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



Exercise 43

Identify the point $(x = 10, y = 54)$.

Side note: in both cases, we assume an abstraction that works with "contiguous areas", not "discrete points" $\Rightarrow$ interval domain, see later.

Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

Various abstract domains

- We have looked at some basic ideas of data abstraction.
- We have looked at program abstraction.

Various abstract domains

- We have looked at some basic ideas of data abstraction.
- We have looked at program abstraction.
- We will now get back to data abstraction, to understand how abstract domains are set up and give a couple of examples of abstract domains.

Various abstract domains

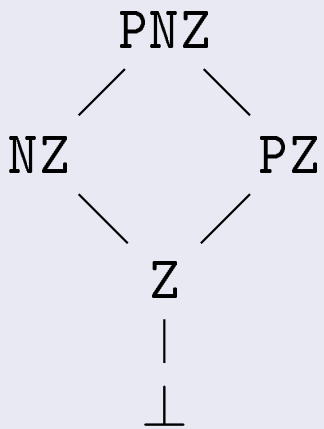
- We have looked at some basic ideas of data abstraction.
- We have looked at program abstraction.
- We will now get back to data abstraction, to understand how abstract domains are set up and give a couple of examples of abstract domains.
- The presentation above suggests that concrete data comes first, one defines an abstraction and then a concretisation. But actually the setup an abstract domain works the other way round:
  - define what are the abstract objects;
  - specify the concretisation γ.
  - The definition of α results from this:

$$\alpha(o) = \min\{o^{\#} \mid o \subseteq \gamma(o^{\#})\}$$

Domain of signs

Definition

Lattice for $\mathcal{D}^\sharp = \{\mathtt{PNZ}, \mathtt{NZ}, \mathtt{PZ}, \mathtt{Z}, \bot\}$:

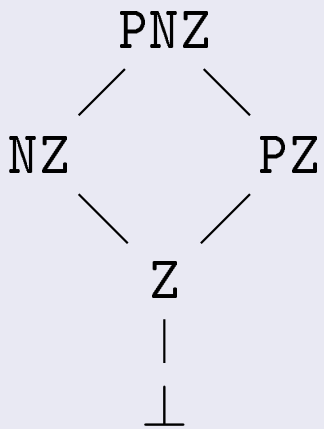


$$
\begin{aligned}
\gamma(\mathtt{PNZ}) &= \mathbb{Z} \\
\gamma(\mathtt{NZ}) &= (-\infty, 0] \\
\gamma(\mathtt{PZ}) &= [0, +\infty) \\
\gamma(\mathtt{Z}) &= \{0\} \\
\gamma(\bot) &= \emptyset
\end{aligned}
$$

Domain of signs

Definition

Lattice for $\mathcal{D}^\sharp = \{\mathtt{PNZ}, \mathtt{NZ}, \mathtt{PZ}, \mathtt{Z}, \bot\}$:



$$\begin{aligned}\gamma(\mathtt{PNZ}) &= \mathbb{Z}\\ \gamma(\mathtt{NZ}) &= (-\infty, 0]\\ \gamma(\mathtt{PZ}) &= [0, +\infty)\\ \gamma(\mathtt{Z}) &= \{0\}\\ \gamma(\bot) &= \emptyset\end{aligned}$$

We define:

$$\alpha(S) = \begin{cases} \mathtt{PNZ} & \text{if } \exists s, s' \in S, s < 0, s' > 0\\ \mathtt{NZ} & \text{if } \forall s \in S, s \leq 0 \wedge \exists s \in S, s < 0\\ \mathtt{PZ} & \text{if } \forall s \in S, s \geq 0 \wedge \exists s \in S, s > 0\\ \mathtt{Z} & \text{if } S = \{0\}\\ \bot & \text{if } S = \emptyset \end{cases}$$

Domain of signs: abstract arithmetic operations

Recall: $\mathit{unary}^\sharp(x) = \alpha(\mathit{unary}(\gamma(x)))$, $\mathit{binary}^\sharp(x, y) = \alpha(\ldots)$.

Domain of signs: abstract arithmetic operations

Recall: $unary^\sharp(x) = \alpha(unary(\gamma(x)))$, $binary^\sharp(x, y) = \alpha(\ldots)$.

- $x^\sharp +^\sharp y^\sharp = \alpha\left(\{x + y \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp)\}\right) =$

$+^\sharp$	PNZ	NZ	PZ	Z	$\bot$
PNZ	PNZ	PNZ	PNZ	PNZ	$\bot$
NZ	PNZ	NZ	PNZ	NZ	$\bot$
PZ	PNZ	PNZ	PZ	PZ	$\bot$
Z	PNZ	NZ	PZ	Z	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

- ...

Domain of signs: abstract arithmetic operations (2)

Exercise 44

$-^\sharp$	PNZ	NZ	PZ	Z	$\bot$
PNZ					
NZ					
PZ					
Z					
$\bot$					

Domain of signs: abstract arithmetic operations with errors

What about posible arithmetic errors, specifically, division by 0?
Recall that in the concrete, errors lead to absence of results,
e.g. $\{10, 20\}/\{-2, 0, 2\} = \{-10, -5, 5, 10\}$. Hence abstract division will be

Exercise

Exercise:

$/^\sharp$	PNZ	NZ	PZ	Z	$\bot$
PNZ					
NZ					
PZ					
Z					
$\bot$					

Domain of signs: abstract arithmetic operations with errors

What about posible arithmetic errors, specifically, division by 0?
Recall that in the concrete, errors lead to absence of results,
e.g. $\{10, 20\}/\{-2, 0, 2\} = \{-10, -5, 5, 10\}$. Hence abstract division will be

Exercise

Exercise:

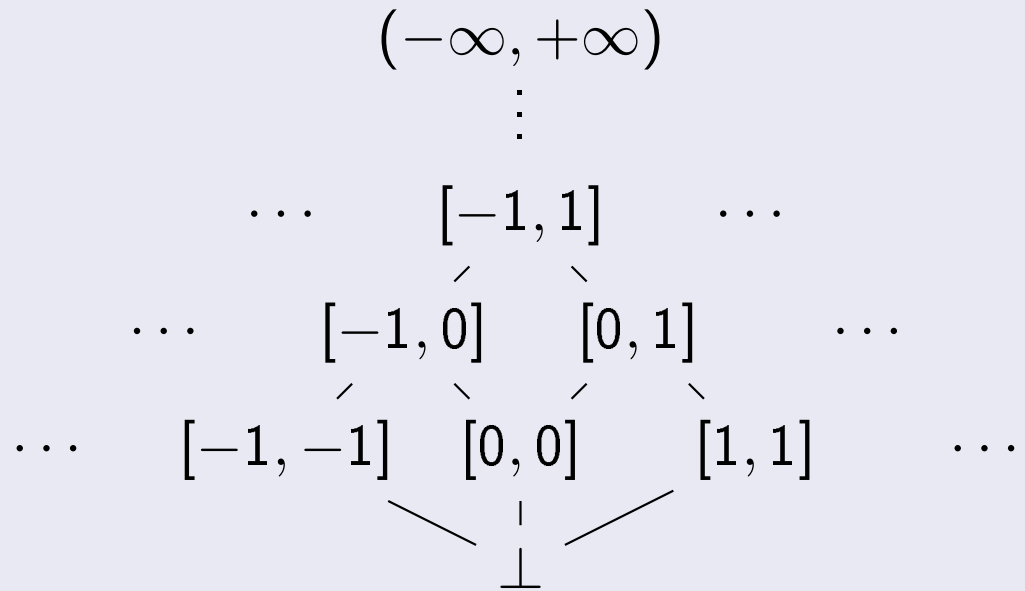
$/^\sharp$	PNZ	NZ	PZ	Z	$\bot$
PNZ	PNZ	PNZ	PNZ	$\bot$	$\bot$
NZ	PNZ	PZ	NZ	$\bot$	$\bot$
PZ	PNZ	NZ	PZ	$\bot$	$\bot$
Z	Z	Z	Z	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Hence, if we infer abstract value PZ for variable x at a certain point, it means that x is definitely non-negative provided we reach that program point; an error might have occurred before.

Domain of intervals

Definition

Lattice of intervals ($\mathcal{D}^\sharp = \{[n_1, n_2] \mid n_1, n_2 \in \mathbb{Z}, n_1 \leq n_2\} \cup \{[n_1, \infty) \mid n_1 \in \mathbb{Z}\} \cup \{(-\infty, n_2] \mid n_2 \in \mathbb{Z}\} \cup \{(-\infty, \infty), \bot\}$)



$\gamma((-\infty, +\infty)) = (-\infty, +\infty)$ $\quad$ $\gamma([n_1, n_2]) = [n_1, n_2]$

$\gamma((-\infty, n]) = (-\infty, n]$ $\quad$ $\gamma(\bot) = \emptyset$

$\gamma([n, +\infty)) = [n, +\infty)$

Domain of intervals: abstraction

We define:

$$\alpha(S) = \begin{cases} [n_1, n_2] & \text{where } n_1 = \min S \text{ and } n_2 = \max S \\ \bot & \text{if } S = \emptyset \end{cases}$$

Same principle as in previous examples: a set S of integers is abstracted as the smallest (w.r.t. $\sqsubseteq$) element in the abstract domain, i.e., the tightest interval, whose concretisation contains S.
We lose information but we want to lose as little as possible, given an abstract domain.

Domain of intervals: Abstract operations

- $x^\sharp +^\sharp y^\sharp = \alpha\left(\{x + y \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp)\}\right) =$

$$\begin{cases} [a + c, b + d] & \text{where } x^\sharp = [a, b] \text{ and } y^\sharp = [c, d] \\ \bot & \text{if } x^\sharp = \bot \text{ or } y^\sharp = \bot \end{cases}$$

- **Exercise 45**

  Subtraction, multiplication

Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

Reminder: The Toy Program

$$
\begin{array}{l}
{}_0\mathrm{x} = \mathbf{rand}(0,\ 12);{}_1\mathrm{y} = 42; \\
\mathbf{while}\ {}_2(\mathrm{x} > 0)\ \{ \\
\qquad {}_3\mathrm{x} = \mathrm{x} - 2; \\
\qquad {}_4\mathrm{y} = \mathrm{y} + 4 \\
\}_5
\end{array}
$$

Executing the Program Using the Interval Domain

	x	y
0:	$(-\infty, \infty)$	$(-\infty, \infty)$
1:	$[0, 12]$	$(-\infty, \infty)$
2*:	$[0, 12]$	$[42, 42]$
3:	$[1, 12]$	$[42, 42]$
4:	$[-1, 10]$	$[42, 42]$
2*:	$[-1, 10]$	$[46, 46]$
3:	$[1, 10]$	$[46, 46]$
4:	$[-1, 8]$	$[46, 46]$

	x	y
2*:	$[-1, 8]$	$[50, 50]$
3:	$[1, 8]$	$[50, 50]$
4:	$[-1, 6]$	$[50, 50]$
2*:	$[-1, 6]$	$[54, 54]$
3:	$[1, 6]$	$[54, 54]$
4:	$[-1, 4]$	$[54, 54]$

	x	y
2*:	$[-1, 4]$	$[58, 58]$
3:	$[1, 4]$	$[58, 58]$
4:	$[-1, 2]$	$[58, 58]$
2*:	$[-1, 2]$	$[62, 62]$
3:	$[1, 2]$	$[62, 62]$
4:	$[-1, 0]$	$[62, 62]$
2:	$[-1, 0]$	$[66, 66]$
5:	$[-1, 0]$	$[66, 66]$

*: quitting the loop is also possible since values ≤ 0 are included in the interval for x, leading to possibilities $\boxed{5:\ [0, 0]\ [42, 42]}$, $\boxed{5:\ [-1, 0]\ [46, 46]}$, $\boxed{5:\ [-1, 0]\ [50, 50]}$, $\boxed{5:\ [-1, 0]\ [54, 54]}$, $\boxed{5:\ [-1, 0]\ [58, 58]}$, $\boxed{5:\ [-1, 0]\ [62, 62]}$, so in summary $\boxed{5:\ [-1, 0]\ [42, 66]}$.

Information Extracted from Execution

- For program point 2, we have to take the union of all points marked "2" above, and thus infer $x = [-1, 12]$, $y = [42, 66]$ (see slide 105).
- We infer this information much more cheaply than by executing in the concrete.
- Especially for an infinite abstract domain like the interval domain, this may still be too expensive and one might consider additional abstraction techniques.
- On the other hand, one might consider refinement techniques.
- This abstract execution just gives an idea of the principle.

Plan

1. Introductory Remarks
2. Timed Automata
3. Abstract Interpretation
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C

Frama-C, a Collection of Tools

The development of Frama-C originates around 1990.

Frama-C, a Collection of Tools

The development of Frama-C originates around 1990.
Several tools inside a single platform

- tools provided as plug-ins
  - 21 plug-ins in the open source distribution
  - outside open source plug-ins
  - closed source plug-ins, either at CEA (about 20) or outside
- plug-ins connected to a kernel
  - provides a uniform setting
  - provides general services
  - synthesizes useful information

Frama-C, a Development Platform

- developed in OCaml ($\approx$ 180 kloc in the open source distribution, $\approx$ 300 kloc with proprietary extensions)
- library dedicated to analysis of C code; development of plug-ins by third party
- powerful low-cost analyser
- Here: EVA for abstract interpretation.

ACSL: Introduction

- ACSL = ANSI/ISO C Specification Language
- First-order logic.

ACSL: Introduction

- ACSL = ANSI/ISO C Specification Language
- First-order logic.
- Specification of a function states pre-conditions and post-conditions.

ACSL: Simple example

```
/*@ requires \valid(a) && \valid(b);
    requires \separated(a,b);
    ensures *a == \at(*b,Pre) && *b == \at(*a,Pre);
*/
void swap(int * a, int * b);
```

- `requires` : pre-condition
- `\valid` and `\separatated`: built-in ACSL predicates
- `ensures` and `\at`, `Pre` (entry point of function)

ACSL: Loop invariants

```
int a[10];
/*@
  loop invariant 0 <= i <= 10;
  loop invariant \forall integer j; 0 <= j < i ==> a[j] == j;
*/
for (int i = 0; i < 10; i++) a[i] = i;
```

- `loop invariants` are true for each loop step: on first entry, and must be preserved, except for `goto`, `break`, `continue`.
- Works for `for`, `while`, `do ... while` loops.
- Particularly useful for deductive verification.

ACSL: Loop invariants (2)

Discussion of example:

- Bounds of the index: `i<=10` although test is `i<10`.
- Second invariant states that the i-1 first cells of the array have been initialized. Why true at beginning?

Annotations

In addition to function specifications, ACSL offers the possibility of writing annotations in the code, in the form of assertions, properties that must be true at a given point.

Bibliography I


Rajeev Alur and David L. Dill.
A theory of timed automata.
Theoretical Computer Science, 126(2):183–235, 1994.


Johan Bengtsson and Wang Yi.
Timed automata: Semantics, algorithms and tools.
In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets 2003*, volume 3098 of *LNCS*, pages 87–124. Springer-Verlag, 2004.


P. Cousot and R. Cousot.
Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.
In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

Bibliography II

Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen.
Model-checking real-time control programs – verifying LEGO mindstorms systems using uppaal.
Technical Report RS-99-53, BRICS, 1999.
BRICS Report Series.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski.
Frama-C: A software analysis perspective.
Formal Asp. Comput., 27(3):573–609, 2015.