M2
Vérification et validation, analyse formelle

Jan-Georg Smaus

Université de Toulouse/IRIT

Year 2022/2023

Planning

1. C intro
2. C LTL (JGS)
3. C LTL (JGS)
4. C LTL (JGS)
5. C TLA (JGS)
6. C EventB
7. C EventB
8. C Spin
9. C Spin
10. TP EventB
11. TP EventB
12. TP EventB
13. TP EventB
14. TP PlusCal
15. TP PlusCal
16. TP PlusCal
17. TP PlusCal
18. C TA (JGS)
19. TP TA (JGS)
20. C TA (JGS)
21. TP TA (JGS)
22. TP TA (JGS)
23. C IA (JGS)
24. C IA (JGS)
25. TP IA (JGS)
26. TP IA (JGS)
27. C IA (JGS)
28. TP IA (JGS)

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

Acknowledgements

The slides and exercises of this chapter are based on material by Marie Duflot-Kremer and Stephan Merz, as well as Stefan Leue.

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

Basic Idea of Model Checking

- Analyze the state graph of a given finite system
  - system: algorithm, circuit, protocol, . . .
  - represented by a transition system
- Properties to verify:
  - safety: nothing bad will ever happen
  - liveness: something good will eventually happen
- Main application domains:
  - reactive systems: permanent interaction with environment
  - parallel and distributed algorithms, protocols, controllers

Basic Idea of Model Checking

- Analyze the state graph of a given finite system
  - system: algorithm, circuit, protocol, . . .
  - represented by a transition system
- Properties to verify:
  - safety: nothing bad will ever happen
  - liveness: something good will eventually happen
- Main application domains:
  - reactive systems: permanent interaction with environment
  - parallel and distributed algorithms, protocols, controllers
- Control is more important than data
- Systems are usually composed of several parts

Three Steps of Model Checking

1. Construct system model
   - describe each system component (e.g., process) by a (finite) automaton
   - languages: TLA^+/ PlusCal, Promela, Petri nets, process algebra, . . .
   - possibly: automatic extraction from source code

2. Specification of expected properties by temporal logic (here: LTL).
   Examples:
   - mutual exclusion $\Box\neg(pc[0] = \text{“cs”} \wedge pc[1] = \text{“cs”})$
   - guaranteed response $(pc[0] = \text{“a2”}) \leadsto (pc[0] = \text{“cs”})$

3. Verification
   - “push-button”: automatic verification by model checker
   - failure: examine counter-example to determine why property fails
   - success: property holds for the model
   - memory overflow / timeout: simplify the model

And again, simply, generally and abstractly . . .

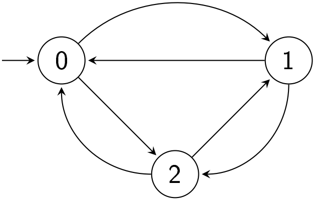
1. System / model
2. Logic
3. Verification (model checking)

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

General Framework for Modelling Discrete Systems

- Transition system ≈ automaton, without acceptance condition
  - example: counter modulo 3



- Generator of runs
  - run: infinite sequence of states and transitions
  - system properties are evaluated over runs
  - flat model: internal structure of states is not represented
    abstract from variables, processes, communication (input/output), . . .
  - observe only which state the system is currently in

Transition systems: definition

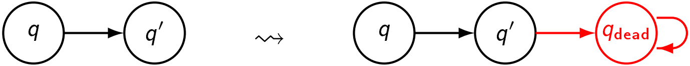
- Abstract model of reactive systems $\mathcal{T} = (Q, I, \delta)$
  - Q finite set of states
  - $I \subseteq Q$ initial states
  - $\delta \subseteq Q \times Q$ (total) transition relation:
    for all $q \in Q$ there exists $q' \in Q$ s.t. $(q, q') \in \delta$

- In practice: $\mathcal{T}$ (i.e., Q and δ) described implicitly
  - TLA^+/Promela: state = assignment of values to state variables
  - Petri nets: state = marking of places in the net

- Size of Q is in general exponential in size of the description of $\mathcal{T}$

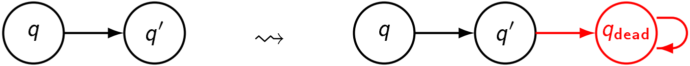
Transition systems: remarks

- Totality of δ
    - technical requirement: simplifies subsequent definitions
    - every finite execution can be extended to an infinite one
    - deadlock must be modelled explicitly



Transition systems: remarks

- Totality of δ
  - technical requirement: simplifies subsequent definitions
  - every finite execution can be extended to an infinite one
  - deadlock must be modelled explicitly

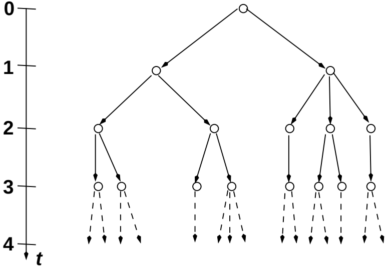


- Variant: labelled transitions $\delta \subseteq Q \times \mathcal{A} \times Q$
  - explicitly identify actions responsible for transitions
  - distinguish internal and communication transitions
  - timed systems, probabilistic systems, ... (more later)

Runs of Transition Systems

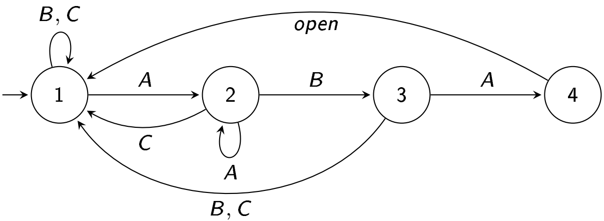
- Run $\rho = q_0 q_1 \ldots$ of $\mathcal{T} = (Q, I, \delta)$
  - $q_0 \in I$ initial state
  - $(q_i, q_{i+1}) \in \delta$ state succession
  - labelled transitions: $\rho = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$

- Unfolding: tree (or forest) representing all runs of $\mathcal{T}$



nodes	states of $\mathcal{T}$
edges	transitions
paths	runs
branching	non-determinism

Example: Digicode as Labelled Transition System



- Door opens in state 4 and is closed otherwise
- The door opens for any code ending in ABA
- Runs of this transition system
    - sequence of states (and actions) describing system evolution
    - $1 \xrightarrow{B} 1 \xrightarrow{A} 2 \xrightarrow{A} 2 \xrightarrow{C} 1 \ldots$
    - $1 \xrightarrow{A} 2 \xrightarrow{B} 3 \xrightarrow{C} 1 \xrightarrow{C} 1 \xrightarrow{A} 2 \xrightarrow{B} 3 \xrightarrow{A} 4 \xrightarrow{open} 1 \ldots$

Exercise 1

Give another run of this system.

Composition of Transition Systems

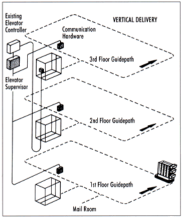
The notion of systems is very abstract and simple. Several features of extension have been proposed.

Composition of Transition Systems

The notion of systems is very abstract and simple. Several features of extension have been proposed.
One of them is composition.

- Systems are usually built from components
  - parallel programs built from processes
  - hardware built from interacting circuits
  - networked systems built from communicating nodes

Example: an elevator is made of a cabin, doors and a controller.



Composition of Transition Systems (2)

- Assemble overall transition system
  - represent each component by separate transition system
  - derive global transition system from component systems
  - different system paradigms reflected by synchronization schemes; here we consider synchronous composition
- Interest for model checking
  - need not explicitly store global transition system
  - component systems can be much smaller than global system
  - can sometimes benefit from symmetries to reduce state space

Synchronizing (Handshake) Composition

- Synchronization via shared actions
  - assume labelled transition systems $\mathcal{T}_i = (Q_i, \mathcal{A}_i, I_i, \delta_i) \quad (i = 1, 2)$
  - synchronized product $\mathcal{T} = (Q_1 \times Q_2, \mathcal{A}_1 \cup \mathcal{A}_2, I_1 \times I_2, \delta)$

$$((q_1, q_2), a, (q_1', q_2')) \in \delta \quad \text{iff}$$

$$\begin{array}{l} a \in \mathcal{A}_1 \setminus \mathcal{A}_2 \;\text{ and }\; (q_1, a, q_1') \in \delta_1 \;\text{ and }\; q_2' = q_2 \quad \text{or} \\ a \in \mathcal{A}_2 \setminus \mathcal{A}_1 \;\text{ and }\; (q_2, a, q_2') \in \delta_2 \;\text{ and }\; q_1' = q_1 \quad \text{or} \\ a \in \mathcal{A}_1 \cap \mathcal{A}_2 \;\text{ and }\; (q_1, a, q_1') \in \delta_1 \;\text{ and }\; (q_2, a, q_2') \in \delta_2 \end{array}$$

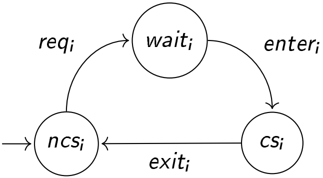
  - joint actions must be executed together, local actions interleave

- Generalizations beyond 2 components
  - multi-party synchronization: actions shared by several components
  - synchronization of two components, the others stutter

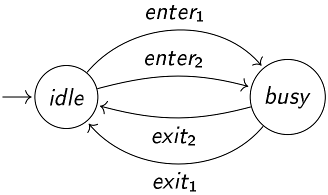
Example: Mutual Exclusion by Joint Actions

Two processes and a controller

Process P_i $(i = 1, 2)$



Controller C

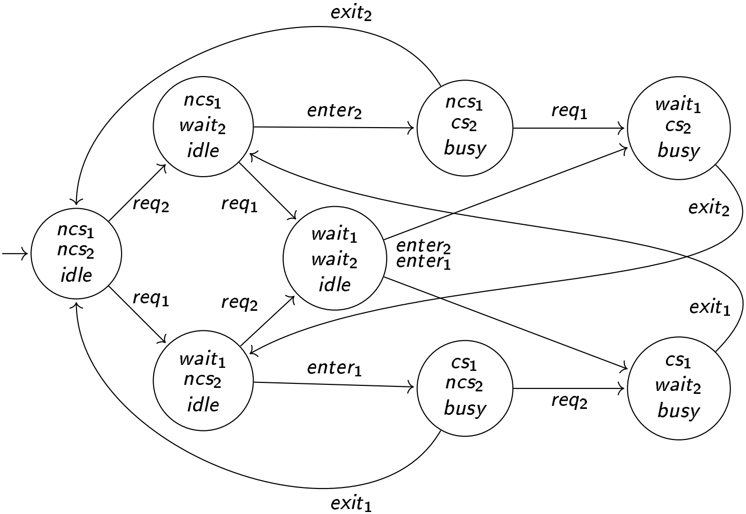


$\mathcal{A}_{P_i} = \{req_i, enter_i, exit_i\}$

$\mathcal{A}_C = \{enter_1, enter_2, exit_1, exit_2\}$

- req_i : local to process P_i
- $enter_i$, $exit_i$: shared between process P_i and controller

Synchronized Product (reachable states)



Exercise 2

How many unreachable states are there?

From simple to complicated and back

- We have started from plain transition systems: very simple formalism, but system descriptions will be huge and hard to read.

From simple to complicated and back

- We have started from plain transition systems: very simple formalism, but system descriptions will be huge and hard to read.
- We considered an extension to have more concise and readable system descriptions: labels (for composition).
  There are other extensions.
  The extension can be “compiled away”, i.e., one can translate an extended system into the simple formalism.

From simple to complicated and back

- We have started from plain transition systems: very simple formalism, but system descriptions will be huge and hard to read.
- We considered an extension to have more concise and readable system descriptions: labels (for composition).
  There are other extensions.
  The extension can be "compiled away", i.e., one can translate an extended system into the simple formalism.
- Now that we know that this can be done, we want to get back to an abstract, theoretical view and hence the initial simple formalism . . .

Kripke Structures

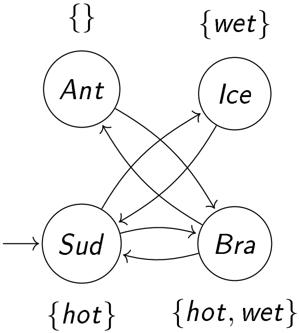
- Add (Boolean) “observations” to the states of a transition system

- Transition system + propositions $\mathcal{K} = (Q, I, \delta, \mathcal{V}, \lambda)$
  - $\mathcal{V}$ set of elementary (“atomic”) propositions
  - $\lambda : Q \to 2^{\mathcal{V}}$ $\lambda(q)$ indicates which propositions are true at q

- Atomic propositions
  - “building blocks” for expressing system properties
  - evaluated at states: v is true at q if $v \in \lambda(q)$, false otherwise
  - examples: – the door protected by the digicode is open
    – the counter value is at least 3
    – process 0 is at the critical section

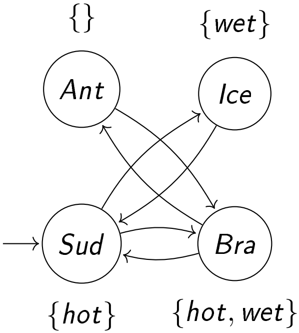
Example of a Kripke Structure

$Q = \{Antarctica, Brazil, Iceland, Sudan\}$.
$I = \{Sudan\}$.
δ as pictured (think of it as "reachability by direct flight").
$\mathcal{V} = \{hot, wet\}$, λ as pictured.



Example of a Kripke Structure

$Q = \{Antarctica, Brazil, Iceland, Sudan\}$.
$I = \{Sudan\}$.
δ as pictured (think of it as "reachability by direct flight").
$\mathcal{V} = \{hot, wet\}$, λ as pictured.



Every run $q_0 q_1 \ldots$ corresponds to an ω-word $\lambda(q_0)\lambda(q_1)\ldots$ over the alphabet $2^{\mathcal{V}}$.
E.g. $Sud\ Bra\ Ant\ Bra\ Sud\ Ice \ldots$ corresponds to
$\{hot\}\{hot, wet\}\{\}\{hot, wet\}\{hot\}\{wet\}\ldots$.

Exercise 3

Give another example.

Where Are We?

1. System / model
2. Logic
3. Verification (model checking)

We have looked at the first point.
We will now look at logic: first, a revision section; then, a section on LTL.

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

Définition formelle de la Lprop (1)

Définition : Soit *PROP* un ensemble de *variables propositionnelles* (typiquement $\{p, q, \ldots\}$).

Définition formelle de la Lprop (1)

Définition : Soit *PROP* un ensemble de *variables propositionnelles* (typiquement $\{p, q, \ldots\}$).
On définit l'ensemble *FORM* des formules par *induction*. *FORM* est le plus petit ensemble qui satisfait les conditions:

1. *Variable propositionnelle:* Si $p \in PROP$, alors $p \in FORM$
2. *Constante "faux":* $\perp \in FORM$

Définition formelle de la Lprop (1)

Définition : Soit *PROP* un ensemble de *variables propositionnelles* (typiquement $\{p, q, \ldots\}$).
On définit l'ensemble *FORM* des formules par *induction*. *FORM* est le plus petit ensemble qui satisfait les conditions:

1. *Variable propositionnelle:* Si $p \in PROP$, alors $p \in FORM$
2. *Constante "faux":* $\perp \in FORM$
3. *Négation:* Si $A \in FORM$, alors $(\neg A) \in FORM$
4. *Conjonction* ("et"): Si $A \in FORM$ et $B \in FORM$, alors $(A \wedge B) \in FORM$
5. *Disjonction* ("ou"): Si $A \in FORM$ et $B \in FORM$, alors $(A \vee B) \in FORM$
6. *Implication* ("si ... alors"): Si $A \in FORM$ et $B \in FORM$, alors $(A \longrightarrow B) \in FORM$

(A et B représentent des formules, c.-à-d., A et B sont des *métavariables*).

Conventions syntaxiques (1)

On peut omettre des parenthèses selon les conventions suivantes:

- *Priorité:* La priorité décroissante des opérateurs est $\neg, \wedge, \vee, \longrightarrow$.
  Exemples:
  - $(A \wedge B \vee C)$ est $((A \wedge B) \vee C)$
  - $(\neg A \vee B)$ est $((\neg A) \vee B)$
  - $(A \wedge B \longrightarrow A \vee B)$ est $((A \wedge B) \longrightarrow (A \vee B))$
- *Associativité:* Les opérateurs binaires associent *à droite*:
  - $A \wedge B \wedge C$ correspond à $(A \wedge (B \wedge C))$
  - $A \longrightarrow B \longrightarrow C$ correspond à $(A \longrightarrow (B \longrightarrow C))$

  Attention: $(A \longrightarrow (B \longrightarrow C))$ et $((A \longrightarrow B) \longrightarrow C)$ ont une sémantique (à voir ultérieurement) différente!
- Enfin, on omet les parenthèses tout autour de la formule (lorsqu'elle n'est pas une variable ou $\bot$).

Conventions syntaxiques (2)

On peut introduire d'autres connecteurs comme *abréviations:*

- *Constante "vrai":* $\top$ défini par $\neg\bot$
- *Double-implication :* $(A \leftrightarrow B)$ défini par $((A \longrightarrow B) \wedge (B \longrightarrow A))$
- *Ou exclusif:* $(A \oplus B)$ défini par $((A \vee B) \wedge (\neg(A \wedge B)))$

Conventions syntaxiques (2)

On peut introduire d'autres connecteurs comme *abréviations:*

- *Constante "vrai":* $\top$ défini par $\neg\bot$
- *Double-implication :* $(A \leftrightarrow B)$ défini par $((A \longrightarrow B) \wedge (B \longrightarrow A))$
- *Ou exclusif:* $(A \oplus B)$ défini par $((A \vee B) \wedge (\neg(A \wedge B)))$

Plus radicalement, on aurait pu définir une syntax minimale avec les connecteurs $\longrightarrow$ et $\neg$, et considérer $\wedge$, $\vee$ comme des abréviations :

- $A \wedge B$ défini par $\neg(A \longrightarrow \neg B)$
- $A \vee B$ défini par $\neg A \longrightarrow B$

Il y a des avantages et inconvenients ...

Conventions syntaxiques (2)

On peut introduire d'autres connecteurs comme *abréviations:*

- *Constante "vrai":* $\top$ défini par $\neg\bot$
- *Double-implication :* $(A \leftrightarrow B)$ défini par $((A \longrightarrow B) \wedge (B \longrightarrow A))$
- *Ou exclusif:* $(A \oplus B)$ défini par $((A \vee B) \wedge (\neg(A \wedge B)))$

Plus radicalement, on aurait pu définir une syntax minimale avec les connecteurs $\longrightarrow$ et $\neg$, et considérer $\wedge$, $\vee$ comme des abréviations :

- $A \wedge B$ défini par $\neg(A \longrightarrow \neg B)$
- $A \vee B$ défini par $\neg A \longrightarrow B$

Il y a des avantages et inconvenients . . .
Aussi, le connecteur $\neg$ est parfois défini comme une abréviation: $\neg A$ défini par $A \longrightarrow \bot$.

Valuation

- En général, une formule propositionnelle n'est ni vraie ni fausse a priori. Cela dépend du contexte, plus précisément, des *variables* de la formule.
- Les contextes possibles pour une formule logique s'appellent des *valuations*.
- Elles décrivent quelle valeur de vérité est associée à chaque variable propositionnelle.

Valuation

- En général, une formule propositionnelle n'est ni vraie ni fausse a priori. Cela dépend du contexte, plus précisément, des *variables* de la formule.
- Les contextes possibles pour une formule logique s'appellent des *valuations*.
- Elles décrivent quelle valeur de vérité est associée à chaque variable propositionnelle.
- *Formellement :* une valuation est une fonction $v : PROP \Rightarrow \{0, 1\}$
- Arbitrairement on pose : $v(p) = 0$: "p faux"; $v(p) = 1$: "p vrai".

Modèle

Chaque valuation peut être étendue de *PROP* à *FORM* de la manière suivante:

1. *(Variable):* $I_v(p) = v(p)$
2. *(Constante):* $I_v(\perp) = 0$
3. *(Négation):* $I_v(\neg A) = 1 - I_v(A)$
4. *(Conjonction):* $I_v(A \wedge B) = min(I_v(A), I_v(B))$
5. *(Disjonction):* $I_v(A \vee B) = max(I_v(A), I_v(B))$
6. *(Implication):* $I_v(A \longrightarrow B) = max(1 - I_v(A), I_v(B))$

On parle alors d'un *modèle* (interprétation).

La notion de “modèle”

Avertissement très important :

- En logique (en particulier dans les UEs de logique en licence à l’UPS), “modèle” veut dire “*une interprétation qui rend (une formule donnée) vraie*” !

La notion de “modèle”

Avertissement très important :

- En logique (en particulier dans les UEs de logique en licence à l’UPS), “modèle” veut dire “*une interprétation qui rend (une formule donnée) vraie*” !
- Ici, dans le contexte du model checking, “modèle” veut dire “*une interprétation*” tout court, qu’elle rende une formule vraie ou fausse !

Modèles et tables de vérité

Table de vérité (TdV) des connecteurs :

A	B	$A \wedge B$	$A \vee B$	$A \longrightarrow B$	$A \leftrightarrow B$	$A \oplus B$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

La table de vérité recense toutes les valuations possibles.
Chaque *ligne* de la TdV correspond à une *valuation* (et donc un modèle). Il y en a $2^2 = 4$.

Modèles et tables de vérité

TdV de $(p \land q) \lor (\neg r)$

p	q	r	$p \land q$	$\neg r$	$(p \land q) \lor (\neg r)$
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

La table de vérité à 2^n lignes et recense toutes les valuations (et donc : modèles) possibles pour n variables propositionnelles.
Chaque *ligne* de la TdV correspond à une *valuation*.

Modèles et tables de vérité

TdV de $(p \wedge q) \vee (\neg r)$

p	q	r	$p \wedge q$	$\neg r$	$(p \wedge q) \vee (\neg r)$
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Exemple: valuation v avec: $v(p) = 1, v(q) = 0, v(r) = 0$
$I_v((p \wedge q) \vee (\neg r)) = max(I_v(p \wedge q), I_v(\neg r)) =$
$max(min(I_v(p), I_v(q)), 1 - I_v(r)) = max(min(1, 0), 1 - 0) = max(0, 1) = 1$

Modèles et tables de vérité

Considérons un sommaire de 3 tables de vérités :

p	q	r	$p \wedge q \longrightarrow q \vee r$	$p \vee q \longrightarrow \neg p \wedge \neg q$	$(p \wedge q) \vee (\neg r)$
0	0	0	1	1	1
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	0	1

La première colonne présente un cas particulièrement intéressant : pour toute valuation v, on a $I_v(A) = 1$ (que des 1 dans sa TdV). Une telle formule est dite *valide* (une *tautologie*).

Exercices

Exercise 4

Déterminez si ces formules sont valides à leurs tables de vérité :

1. $(p \wedge q) \longrightarrow p$
2. $(p \longrightarrow r) \longrightarrow (p \vee q \longrightarrow r)$
3. $(p \wedge (q \longrightarrow r)) \longrightarrow ((\neg p \vee q) \longrightarrow (p \wedge r))$

Conséquence (1)

Définition: Soient $H_1, \ldots, H_n, C \in \mathit{FORM}$.
On dit que C est une *conséquence (logique)* de $H_1, \ldots, H_n$, écrit $\{H_1, \ldots, H_n\} \models C$, ssi tout modèle vrai de $\{H_1, \ldots, H_n\}$ est aussi un modèle vrai de C.

Cas particulier: On écrit $\models C$ au lieu de $\{\} \models C$ (c.-à-d., C est valide).

Conséquence (1)

Définition: Soient $H_1, \ldots, H_n, C \in FORM$.
On dit que C est une *conséquence (logique)* de $H_1, \ldots, H_n$, écrit $\{H_1, \ldots, H_n\} \models C$, ssi tout modèle vrai de $\{H_1, \ldots, H_n\}$ est aussi un modèle vrai de C.

Cas particulier: On écrit $\models C$ au lieu de $\{\} \models C$ (c.-à-d., C est valide).

Nous verrons plus tard des définitions alternatives de $\models$ qui sont techniquement différentes, mais parfaitement cohérentes avec celle-ci. Moralement, l'idée est toujours la suivante :

- À gauche du $\models$, on fixe un cadre/monde/(ensemble de) modèle(s).
- La formule à droite du $\models$ est vraie dans ce monde (...).

En particulier, dans la logique propositionnelle, le monde ne bouge pas. Plus tard, on va l'enrichir pour parler de ce qui est vrai maintenant, demain, ...

Conséquence (2)

p	q	r	$p \wedge q$	$\neg r$	$(p \wedge q) \vee (\neg r)$
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Ici, on peut observer que $p, \neg r \models (p \wedge q) \vee (\neg r)$: chaque fois que *toutes* les hypothèses sont vérifiées, la conclusion l'est aussi.

Exercices

Exercise 5

Vrai ou faux?

- $\{p, q\} \models p \vee q$
- $\{p, q\} \models p \wedge q$
- $\{p, q\} \models p \longrightarrow q$
- $\{p \longrightarrow q, q\} \models p$
- $\{p \longrightarrow q, p\} \models q$
- $\{p \longrightarrow q, \neg q\} \models \neg p$

Logique des prédicats : Termes

Termes: Soit

- VAR un ensemble de variables d'individus
- Pour quelque(s) $n \in \mathbb{N}$, FON_n un ensemble des fonctions n-aires

L'ensemble $TERM$ des termes est défini comme le plus petit ensemble qui satisfait:

1. *Variable:* si $x \in VAR$, alors $x \in TERM$
2. *Application de fonction:* si $t_1 \in TERM, \ldots, t_n \in TERM$ et $f \in FON_n$, alors $f(t_1, \ldots, t_n) \in TERM$

On appelle des éléments de FON_0 des *constantes*.
Souvent, on écrit c au lieu de $c()$.
Exemples: Soient $f \in FON_2$, $x, y \in VAR$, $g \in FON_1$, $\pi \in FON_0$

- $f(x, \pi) \in TERM$
- $f(x, g(f(y, \pi))) \in TERM$

Logique des prédicats : Formules

Formules: Soit $PRED_n$ un ensemble des prédicats n-aires

L'ensemble $FORM$ des formules est défini comme le plus petit ensemble qui satisfait:

1. *Application de prédicat:* si $t_1 \in TERM, \dots, t_n \in TERM$ et $P \in PRED_n$, alors $P(t_1, \dots, t_n) \in FORM$
2. *Constante "faux":* $\perp \in FORM$
3. *Négation:* Si $A \in FORM$, alors $(\neg A) \in FORM$
4. *Connecteurs binaires:* Si $A \in FORM$ et $B \in FORM$, alors $(A \wedge B) \in FORM, (A \vee B) \in FORM, (A \longrightarrow B) \in FORM$
5. *Quantificateur universel* ("pour tout"): Si $A \in FORM$ et $x \in VAR$, alors $(\forall x.A) \in FORM$
6. *Quantificateur existentiel* ("il existe"): Si $A \in FORM$ et $x \in VAR$, alors $(\exists x.A) \in FORM$

À noter: $FORM$ dépend de $TERM$, mais pas inversement.

Logique des prédicats : Sémantique

- En général, la sémantique de la logique des prédicats est lourde à présenter.
- Pourtant elle est bien maîtrisée par les informaticiens (théoriques), et dans les articles scientifiques on trouve souvent des mots comme “sémantique évidente”.

Logique des prédicats : Sémantique

- En général, la sémantique de la logique des prédicats est lourde à présenter.
- Pourtant elle est bien maîtrisée par les informaticiens (théoriques), et dans les articles scientifiques on trouve souvent des mots comme "sémantique évidente".
- Évident ou non, ici nous voulons nous concentrer sur les aspects temporels, et la *sémantique* de la logique des prédicats classique *ne peut pas être un problème* pour nous.

Logique des prédicats : Sémantique

- En général, la sémantique de la logique des prédicats est lourde à présenter.
- Pourtant elle est bien maîtrisée par les informaticiens (théoriques), et dans les articles scientifiques on trouve souvent des mots comme "sémantique évidente".
- Évident ou non, ici nous voulons nous concentrer sur les aspects temporels, et la *sémantique* de la logique des prédicats classique *ne peut pas être un problème* pour nous.
- Vrai ou faux ?

$$\begin{array}{lll} 1+1=2 & \exists x.2\cdot x=4 & \forall x.\exists y.y>x \\ 1+1=3 & \exists x.2\cdot x=3 & 2\in\{1,4\} \end{array}$$

Where Are We?

1. System / model
2. Logic
3. Verification (model checking)

We have revised some notions of logic in general. Now, we will look at LTL, the logic of interest for model checking.

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

Formal Description of Properties

- Need a language for expressing properties of systems
  - system under verification represented as a transition system
  - properties of systems should be expressed unambiguously
- Natural language is ambiguous

Example

"Every student takes a computer science lecture."

- There is a CS lecture taken by all students.
- For every student *s* there is a CS lecture that *s* takes.

No obvious interpretation a priori!

- Mathematical logic allows formalizing such statements

Temporal Properties

- Wish to express properties of system executions
  - After the emergency brake is pulled, the train will stop.
  - After subscribing to a phone service, users may receive calls.
  - When the window is broken, an alarm will sound until it is switched off.
  - The lift does not move unless somebody previously requested it.

- Properties on succession of states / events
  - something holds { before / after / between } some other event(s)
  - no references to absolute time (for the moment)

Temporal Properties in First-Order Logic

- Add explicit time parameter
  - propositions can be true or false at different time points t
  - relate different time points (e.g., $t+1$, $t' \geq t$, $\ldots$)
- Example
  - After the emergency brake is pulled, the train will stop.

    $$\forall t : \mathit{Brake}(t) \longrightarrow \exists t' : t' \geq t \land \mathit{Stop}(t')$$
  - When the window is broken, an alarm will sound until it is switched off.

    $$\forall t : \mathit{Break}(t) \longrightarrow \exists t' \geq t : \mathit{Off}(t') \land \forall t'' : t \leq t'' < t' \longrightarrow \mathit{Alarm}(t'')$$
- Possible, but somewhat clumsy
  - especially for properties that contain several temporal references
  - moreover, reasoning in first-order logic is undecidable in general

. . . temporal logics used in verification tend to be decidable

Temporal Properties in First-Order Logic (2)

- The train passes through stations A, then B, then C

$$\exists t, t', t'' : t < t' \wedge t' < t'' \wedge AtA(t) \wedge AtB(t') \wedge AtC(t'')$$

- When the window is broken, the alarm sounds until switched off

$$\begin{aligned} \forall t : Break(t) \longrightarrow \exists t' : t' > t \wedge Off(t') \wedge \\ \forall t'' : t \leq t'' \wedge t'' < t' \longrightarrow Alarm(t'') \end{aligned}$$

- The lift does not move unless there was a previous request

$$\forall t : Move(t) \longrightarrow \exists t' : t' \leq t \wedge Request(t')$$

  - reference to past vs. future time points

- After subscribing to a phone service, users may receive calls

$$\forall t : Subscribe(t) \longrightarrow \exists t' : t' > t \wedge Receive(t')$$

Temporal Properties in First-Order Logic (2)

- The train passes through stations A, then B, then C

$$\exists t, t', t'' : t < t' \wedge t' < t'' \wedge AtA(t) \wedge AtB(t') \wedge AtC(t'')$$

- When the window is broken, the alarm sounds until switched off

$$\begin{aligned}\forall t : Break(t) \longrightarrow \exists t' : t' > t \wedge Off(t') \wedge \\ \forall t'' : t \leq t'' \wedge t'' < t' \longrightarrow Alarm(t'')\end{aligned}$$

- The lift does not move unless there was a previous request

$$\forall t : Move(t) \longrightarrow \exists t' : t' \leq t \wedge Request(t')$$

  - reference to past vs. future time points

- After subscribing to a phone service, users may receive calls

$$\forall t : Subscribe(t) \longrightarrow \exists t' : t' > t \wedge Receive(t')$$

  - the formula says that a call *will* be received!
  - expressing the original property requires quantifying over runs, not just over time points ("branching time")

Do we Need First-order Logic?

- In the previous examples, the power of first-order logic (having predicates with arguments and quantifiers) is only needed to deal with time. Apart from that, propositional logic would be enough.
- Rather than having a model like in propositional logic, we will now see a sequence of models, to capture the change of truth over time.
- We will enrich propositional logic with operators that talk about time.

The Meaning of $\models$

Recall that when you write $\ldots \models \varphi$, the idea is always the following:

- À gauche du $\models$, on fixe un cadre/monde/(ensemble de) modèle(s).
- La formule à droite du $\models$ est vrai dans ce monde $(\ldots)$.

The Meaning of $\models$

Recall that when you write $\ldots \models \varphi$, the idea is always the following:

- À gauche du $\models$, on fixe un cadre/monde/(ensemble de) modèle(s).
- La formule à droite du $\models$ est vrai dans ce monde $(\ldots)$.

In the context of systems that evolve over time, we could have:

- A sequence of sets of atomic propositions, e.g.

$$\{hot\}\{\}\{hot, wet\}\ldots \models$$

- The states of a Kripke structure that have these sets associated:

$$Sud\ Ant\ Bra \ldots \models$$

- The Kripke structure itself:

$$\mathcal{K} \models$$

Linear-Time Temporal Logic: Informally (1)

- Eliminate explicit time parameter
  - formulas are evaluated over infinite state sequences σ
  - they can be true or false at different time points
  - atomic formulas: elementary properties evaluated at states

    $\sigma \models \mathit{Break}$ proposition *Break* is true at initial state of σ
  - if $\sigma = q_0 q_1 \ldots$ is a run of a Kripke structure $\mathcal{K} = (Q, I, \delta, \mathcal{V}, \lambda)$:

    $\sigma \models v$ determined by $\lambda(q_0)$, for $v \in \mathcal{V}$

- Standard Boolean connectives $\wedge, \vee, \neg, \longrightarrow, \leftrightarrow$
  - applied to arbitrary formulas, with standard interpretation
  - $\sigma \models \mathit{Alarm} \wedge \neg \mathit{Off}$ *Alarm* true, but *Off* false at initial state

Linear-Time Temporal Logic: Informally (2)

- Temporal connectives for temporal references
  - change “point of evaluation” of (sub-)formulas

always φ	φ true at all suffixes	$\mathbf{G}\,\varphi$	$(\Box\varphi)$
eventually φ	φ true at some suffix	$\mathbf{F}\,\varphi$	$(\Diamond\varphi)$
next φ	φ true at immediate suffix	$\mathbf{X}\,\varphi$	$(\circ\varphi)$
φ until ψ	φ remains true until ψ becomes true	$\varphi\,\mathbf{U}\,\psi$	

- Examples
  - $\mathbf{G}(\mathit{Brake} \longrightarrow \mathbf{F}\,\mathit{Stop})$
  - $\mathbf{G}(\mathit{Break} \longrightarrow (\mathit{Alarm}\ \mathbf{U}\ \mathit{Off}))$

Formal Syntax of LTL

- LTL: compact syntax for properties of runs
  - formulas evaluated over infinite state sequences
  - system satisfies φ if φ holds of every run
- Inductive definition of LTL formulas

$$\begin{array}{lcll}
\varphi & ::= & v \in \mathcal{V} & \text{atomic formulas} \\
 & | & \neg\varphi, \varphi \vee \varphi & \text{Boolean connectives} \\
 & | & \mathbf{X}\,\varphi & \text{next state } (\circ\varphi) \\
 & | & \varphi\,\mathbf{U}\,\varphi & \text{until } (\omega\ \textbf{until}\ \psi)
\end{array}$$

Exercise 6

How is this notation called?

- Abbreviations
  - $\wedge, \longrightarrow, \leftrightarrow, \mathsf{true}, \mathsf{false}$ as in propositional logic
  - $\mathbf{F}\,\varphi \;\equiv\; \mathsf{true}\,\mathbf{U}\,\varphi$ eventually φ (finally, $\Diamond\varphi$)
  - $\mathbf{G}\,\varphi \;\equiv\; \neg\,\mathbf{F}\,\neg\varphi$ always φ (globally, $\Box\varphi$)

Formal Semantics of LTL

- Formulas φ evaluated over infinite sequences of states
  - atomic formulas interpreted by labelling $\lambda : Q \to 2^{\mathcal{V}}$
  - notations: for $\sigma = q_0 q_1 \ldots$, we denote by $\sigma[n..]$ the suffix $q_n q_{n+1} \ldots$.
- Inductive definition of $\sigma \models \varphi$

$$\begin{array}{lll}
\sigma \models v & \text{iff} & v \in \lambda(\sigma_0) \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models \varphi \vee \psi & \text{iff} & \sigma \models \varphi \ \text{ or } \ \sigma \models \psi \\
\sigma \models \mathbf{X}\,\varphi & \text{iff} & \sigma[1..] \models \varphi \\
\sigma \models \varphi\ \mathbf{U}\ \psi & \text{iff} & \text{there is } k \in \mathbb{N} \text{ such that } \sigma[k..] \models \psi \\
 & & \qquad \text{and } \sigma[i..] \models \varphi \text{ for all } 0 \leq i < k
\end{array}$$

- Semantics of derived temporal connectives
  - $\sigma \models \mathbf{F}\,\varphi$ iff $\sigma[k..] \models \varphi$ for some $k \in \mathbb{N}$
  - $\sigma \models \mathbf{G}\,\varphi$ iff $\sigma[k..] \models \varphi$ for all $k \in \mathbb{N}$

Example: Interpretation of LTL Formulas

Exercise 7

Which of the following formulas are true? (true = 1, false = 0)

hot	0	0	1	1	0	1	1	0	1	⋯	(always 1)
wet	1	1	0	0	0	0	1	1	0	⋯	(always 0)
State:	Ice	Ice	Sud	Sud	Ant	Sud	Bra	Ice	Sud	⋯	(always Sud)

1. $\mathbf{G}(\neg hot \longrightarrow wet)$
2. $\mathbf{F}(hot \wedge \neg wet)$
3. $\neg hot\ \mathbf{U}\ \neg wet$
4. $\mathbf{G}(\neg hot \longrightarrow (\neg hot\ \mathbf{U}\ \neg wet))$
5. $\mathbf{G}\,\mathbf{F}(wet)$
6. $\mathbf{F}\,\mathbf{G}(\neg hot \longrightarrow wet)$
7. $wet\ \mathbf{U}\ hot$
8. $\mathbf{G}(wet\ \mathbf{U}\ hot)$

Example 2: Interpretation of LTL Formulas

Exercise 8

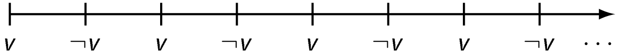
Which of the following formulas are true? (true = 1, false = 0)

hot	0	0	1	1	0	1	1	0	1	⋯	(always 0 1)
wet	1	1	0	0	0	0	1	1	0	⋯	(always 1 0)
State:	Ice	Ice	Sud	Sud	Ant	Sud	Bra	Ice	Sud	⋯	(always Ice Sud)

1. $\mathbf{G}(\neg hot \longrightarrow wet)$
2. $\mathbf{F}(hot \wedge \neg wet)$
3. $\neg hot \ \mathbf{U} \ \neg wet$
4. $\mathbf{G}(\neg hot \longrightarrow (\neg hot \ \mathbf{U} \ \neg wet))$
5. $\mathbf{G}\,\mathbf{F}(wet)$
6. $\mathbf{F}\,\mathbf{G}(\neg hot \longrightarrow wet)$
7. $wet \ \mathbf{U} \ hot$
8. $\mathbf{G}(wet \ \mathbf{U} \ hot)$

Infinitely Often and Persistence

- $\mathbf{G}\,\mathbf{F}\,\varphi$
  - for every suffix there is a subsequent suffix satisfying φ
  - φ is infinitely often true
- $\mathbf{F}\,\mathbf{G}\,\varphi$
  - there is a suffix such that all subsequent suffixes satisfy φ
  - φ is false only finitely often, φ is persistent
- $\mathbf{F}\,\mathbf{G}\,\varphi$ is strictly stronger than $\mathbf{G}\,\mathbf{F}\,\varphi$

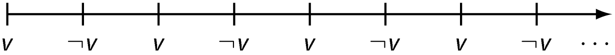


$\models \mathbf{G}\,\mathbf{F}\,v$

$\not\models \mathbf{F}\,\mathbf{G}\,v$

Infinitely Often and Persistence

- $\mathbf{G}\,\mathbf{F}\,\varphi$
  - for every suffix there is a subsequent suffix satisfying φ
  - φ is infinitely often true
- $\mathbf{F}\,\mathbf{G}\,\varphi$
  - there is a suffix such that all subsequent suffixes satisfy φ
  - φ is false only finitely often, φ is persistent
- $\mathbf{F}\,\mathbf{G}\,\varphi$ is strictly stronger than $\mathbf{G}\,\mathbf{F}\,\varphi$



$\models \mathbf{G}\,\mathbf{F}\,v$

$\not\models \mathbf{F}\,\mathbf{G}\,v$

- Combinations
  - $\mathbf{G}(req \longrightarrow \mathbf{F}\,get)$ every request will be satisfied
  - $\mathbf{G}(\mathbf{G}\,req \longrightarrow \mathbf{F}\,get)$ every persistent request will be satisfied
  - $\mathbf{G}(\mathbf{G}\,\mathbf{F}\,req \longrightarrow \mathbf{F}\,get)$ every repeated request will be satisfied

Quelques lois de LTL

- Commutativité de **X** avec tous les opérateurs

$\mathbf{X}\,\neg A \equiv \neg\,\mathbf{X}\,A$

$\mathbf{X}(A \longrightarrow B) \equiv (\mathbf{X}\,A \longrightarrow \mathbf{X}\,B)$

$\mathbf{X}\,\mathbf{F}\,A \equiv \mathbf{F}\,\mathbf{X}\,A$

$\mathbf{X}(A \vee B) \equiv \mathbf{X}\,A \vee \mathbf{X}\,B$

$\mathbf{X}\,\mathbf{G}\,A \equiv \mathbf{G}\,\mathbf{X}\,A$

$\mathbf{X}(A\ \mathbf{U}\ B) \equiv (\mathbf{X}\,A\ \mathbf{U}\ \mathbf{X}\,B)$

- Monotonicité

$\mathbf{G}(A \longrightarrow B) \longrightarrow (\mathbf{X}\,A \longrightarrow \mathbf{X}\,B)$

$\mathbf{G}(A \longrightarrow B) \longrightarrow (\mathbf{G}\,A \longrightarrow \mathbf{G}\,B)$

$\mathbf{G}(A \longrightarrow B) \longrightarrow (\mathbf{F}\,A \longrightarrow \mathbf{F}\,B)$

$\mathbf{G}(A \longrightarrow B) \longrightarrow (A\ \mathbf{U}\ C \longrightarrow B\ \mathbf{U}\ C)$

- Distributivité

$\mathbf{G}(A \wedge B) \equiv \mathbf{G}\,A \wedge \mathbf{G}\,B$

$\mathbf{F}\,\mathbf{G}(A \wedge B) \equiv \mathbf{F}\,\mathbf{G}\,A \wedge \mathbf{F}\,\mathbf{G}\,B$

$\mathbf{F}(A \vee B) \equiv \mathbf{F}\,A \vee \mathbf{F}\,B$

$\mathbf{G}\,\mathbf{F}(A \vee B) \equiv \mathbf{G}\,\mathbf{F}\,A \vee \mathbf{G}\,\mathbf{F}\,B$

Quelques lois de LTL (2)

- Dualité

  $\neg\,\mathbf{G}\,A \equiv \mathbf{F}\,\neg A$ $\qquad$ $\neg\,\mathbf{F}\,A \equiv \mathbf{G}\,\neg A$

  $\neg\,\mathbf{G}\,\mathbf{F}\,A \equiv \mathbf{F}\,\mathbf{G}\,\neg A$ $\qquad$ $\neg\,\mathbf{F}\,\mathbf{G}\,A \equiv \mathbf{G}\,\mathbf{F}\,\neg A$

- Lois récursives

  $\mathbf{G}\,A \equiv A \wedge \mathbf{X}\,\mathbf{G}\,A$ $\qquad$ $\mathbf{F}\,A \equiv A \vee \mathbf{X}\,\mathbf{F}\,A$

  $A\,\mathbf{U}\,B \equiv B \vee (A \wedge \mathbf{X}(A\,\mathbf{U}\,B))$

Exercise 9

Prouver quelques uns de ces lois.

Exercise: Properties of Binary Consensus

Consider a system of N processes $P_1, \dots, P_N$ where the state of P_i is given by a Boolean variable v_i, indicating its *local value*, and d_i, indicating if P_i has *decided* (initially false). The *Consensus problem* consists in arriving at a state where every process has decided and where the local values of all processes are identical. This is expressed by the four following properties.

1. Validity. At any state, any value v_i must equal the initial value of some v_j (i.e., no values other than those initially present are introduced).
2. Irrevocability. Once P_i decides (i.e., sets its variable d_i to true), the variables v_i and d_i never change again.
3. Agreement. Any two processes P_i and P_j that have decided agree on the values of v_i and v_j.
4. Termination. Every process P_i decides eventually.

Exercise 10

Express these four properties by LTL formulas.

Typical Properties in LTL

Safety properties:

- invariants: $\mathbf{G}\, p$ where p is non-temporal

$\mathbf{G}\, \neg(crit_1 \wedge crit_2)$	mutual exclusion
$\mathbf{G}(pre_1 \vee \ldots \vee pre_n)$	deadlock freedom

- formulas built from **X** and **G**

$\mathbf{G}(p \longrightarrow \mathbf{X}\, q)$every p is immediately followed by q

$\mathbf{G}(p \longrightarrow \mathbf{G}\, q)q$ stays true after p

Typical Properties in LTL

Safety properties:

- invariants: $\mathbf{G}\, p$ where p is non-temporal

$\mathbf{G}\, \neg(crit_1 \wedge crit_2)$	mutual exclusion
$\mathbf{G}(pre_1 \vee \ldots \vee pre_n)$	deadlock freedom

- formulas built from **X** and **G**

$\mathbf{G}(p \longrightarrow \mathbf{X}\, q)$every p is immediately followed by q

$\mathbf{G}(p \longrightarrow \mathbf{G}\, q)q$ stays true after p

Liveness properties:

- reply, recurrence $\mathbf{G}(p \longrightarrow \mathbf{F}\, q)$

$\mathbf{G}(try_1 \longrightarrow \mathbf{F}\, crit_1)$	guaranteed access to critical section
$\mathbf{G}(\quad \mathbf{F}\, \neg crit_1)$	finite stay in critical section

- reactivity $\mathbf{G}\,\mathbf{F}\, p \longrightarrow \mathbf{G}\,\mathbf{F}\, q$

$\mathbf{G}\,\mathbf{F}(try_1 \wedge \neg crit_2) \longrightarrow \mathbf{G}\,\mathbf{F}\, crit_1$	(strong) fairness

Typical Properties in LTL (2)

Weak fairness

- if the action is always enabled, it will eventually happen
- $\mathrm{WF}(A) \;\equiv\; \mathbf{G}(\mathbf{G}\, \mathit{enabled}_A \longrightarrow \mathbf{F}\, \mathit{taken}_A)$

Typical Properties in LTL (2)

Weak fairness

- if the action is always enabled, it will eventually happen
- $\mathrm{WF}(A) \ \equiv\ \mathbf{G}(\mathbf{G}\, enabled_A \longrightarrow \mathbf{F}\, taken_A)$

Strong fairness

- if the action is enabled infinitely often, it will eventually happen
- $\mathrm{SF}(A) \ \equiv\ \mathbf{G}(\mathbf{G}\,\mathbf{F}\, enabled_A \longrightarrow \mathbf{F}\, taken_A)$

Other:

- precedence $p_1\ \mathbf{U} \ldots \mathbf{U}\ p_n$

$\mathbf{G}(try_1 \wedge try_2 \longrightarrow \neg crit_2\ \mathbf{U}\ crit_2\ \mathbf{U}\ \neg crit_2\ \mathbf{U}\ crit_1)$ 1-bounded overtaking

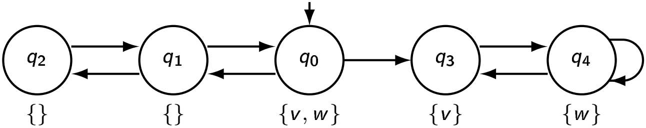
$\mathcal{K}$-Validity (LTL formulas and entire systems)

Given a Kripke structure $\mathcal{K}$, we write $\mathcal{K} \models \varphi$ iff for every run $q_0 q_1 \ldots$ of $\mathcal{K}$, we have $q_0 q_1 \ldots \models \varphi$.

$\mathcal{K}$-Validity (LTL formulas and entire systems)

Given a Kripke structure $\mathcal{K}$, we write $\mathcal{K} \models \varphi$ iff for every run $q_0 q_1 \ldots$ of $\mathcal{K}$, we have $q_0 q_1 \ldots \models \varphi$.

Exercise: Consider the following Kripke structure $\mathcal{K}$.



Exercise 11

Determine if $\mathcal{K} \models \varphi_i$ holds for the following formulas φ_i. Briefly justify your answers.

$$\begin{array}{lll lll}
\varphi_1 & = & \mathbf{G}(v \longrightarrow w \vee \mathbf{X}\, w) & \varphi_6 & = & \mathbf{G}\,\mathbf{F}\, w \\
\varphi_2 & = & \neg w \longrightarrow (\neg w \;\mathbf{U}\; w) & \varphi_7 & = & \mathbf{G}\,\mathbf{F}\, \neg v \\
\varphi_3 & = & \mathbf{G}(\neg w \longrightarrow (\neg w \;\mathbf{U}\; w)) & \varphi_8 & = & \mathbf{G}\,\mathbf{F}\, \neg w \\
\varphi_4 & = & \mathbf{G}\,\mathbf{F}\, v & \varphi_9 & = & (\mathbf{G}\,\mathbf{F}\, v) \longrightarrow (\mathbf{G}\,\mathbf{F}\, w) \\
\varphi_5 & = & \mathbf{G}\,\mathbf{F}(v \wedge w) & \varphi_{10} & = & (\mathbf{G}\,\mathbf{F}(v \wedge w)) \longrightarrow (\mathbf{G}\,\mathbf{F}(v \wedge \neg w))
\end{array}$$

Where Are We?

1. System / model
2. Logic
3. Verification (model checking)

We have looked at LTL, the logic of interest for model checking. Now we study an algorithm for model checking.

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

Example: Verifying a Persistence Property

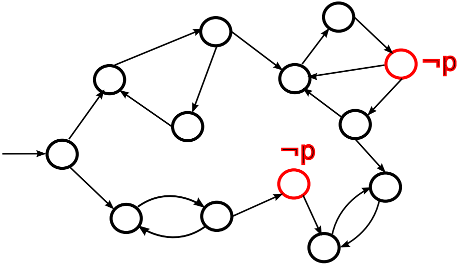
Consider the problem of verifying $\mathcal{K} \models \mathbf{F}\,\mathbf{G}\, v$.

- The property is violated iff there exists a run $\sigma = q_0 q_1 \dots q_{i_0} \dots q_{i_1} \dots q_{i_2} \dots$ such that $v \notin \lambda(q_{i_j})$ for all $j \in \mathbb{N}$. Since $\mathcal{K}$ is finite-state, we must have $q_{i_j} = q_{i_k}$ for some $k > j$.
- The prefix of σ given by $q_0 \Rightarrow^* q_{i_j} \Rightarrow^+ q_{i_k}$ is effectively a “lasso” through a state where v is false

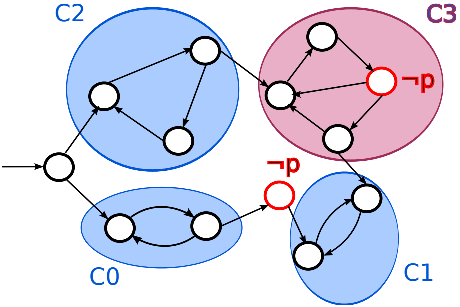
To search for a lasso, we inspect graph G of reachable states of $\mathcal{K}$:

- compute strongly connected components of G (can be done by Tarjan’s algorithm: linear in size of G).
- for each component, check if it contains some q with $v \notin \lambda(q)$. If you find such a component, you have found a lasso. Hence $\mathcal{K} \not\models \mathbf{F}\,\mathbf{G}\, v$.

Example: Verifying **F G** p



Example: Verifying **F G** p



$\mathcal{K} \not\models \mathbf{F}\,\mathbf{G}\,p$: component C3 contains state where p is false.

Exercise 12

Why is the occurrence of $\neg p$ in C3 decisive, why does the other occurrence not matter?

Plan

1. Model Checking
   - Motivation
   - Discrete transition systems
   - Logic Revised
   - Linear Temporal Logic
   - Model checking algorithm
   - TLA
2. Timed Automata
3. Abstract Interpretation and Deductive Verification

TLA: Introduction

- TLA ressemble à LTL, vous allez reconnaître quelques opérateurs temporels.
- TLA est basée sur la logique des prédicats. On peut imaginer une version de LTL basée sur la logique des prédicats, mais il est difficile à imaginer TLA basée sur la logique propositionnelle.
- Dans TLA, vous avez un seul langage pour décrire un système (un programme qui modifie des variables) et exprimer les propriétés désirées de ce système.

TLA: Introduction

- TLA ressemble à LTL, vous allez reconnaître quelques opérateurs temporels.
- TLA est basée sur la logique des prédicats. On peut imaginer une version de LTL basée sur la logique des prédicats, mais il est difficile à imaginer TLA basée sur la logique propositionnelle.
- Dans TLA, vous avez un seul langage pour décrire un système (un programme qui modifie des variables) et exprimer les propriétés désirées de ce système.
- On suppose comme fixés des ensembles de symboles de fonction et des symboles de prédicat.
- On suppose deux ensembles $\mathcal{X}_r$ et $\mathcal{X}_f$ de variables rigides et flexibles.

Formules d'action et formules d'état

Pour $\mathcal{X}_f = \{x, y, \ldots\}$, notons $\mathcal{X}_f' = \{x', y', \ldots\}$.
Une formule d'action est une formule de la logique des prédicats sur des variables dans $\mathcal{X}_r$, $\mathcal{X}_f$ et $\mathcal{X}_f'$.
Intuition: une action modifie les variables flexibles. Les variables sans $'$ se réfèrent aux valeurs avant l'exécution de l'action, et les variables avec $'$ se réfèrent aux valeurs après l'exécution de l'action.

Formules d'action et formules d'état

Pour $\mathcal{X}_f = \{x, y, \ldots\}$, notons $\mathcal{X}'_f = \{x', y', \ldots\}$.
Une formule d'action est une formule de la logique des prédicats sur des variables dans $\mathcal{X}_r$, $\mathcal{X}_f$ et $\mathcal{X}'_f$.
Intuition: une action modifie les variables flexibles. Les variables sans $'$ se réfèrent aux valeurs avant l'exécution de l'action, et les variables avec $'$ se réfèrent aux valeurs après l'exécution de l'action.
Cas spécial : une formule qui n'a aucune occurrence libre d'une variable de $\mathcal{X}'_f$ est appelée formule d'état.

Formules d'action et formules d'état

Pour $\mathcal{X}_f = \{x, y, \ldots\}$, notons $\mathcal{X}'_f = \{x', y', \ldots\}$.
Une formule d'action est une formule de la logique des prédicats sur des variables dans $\mathcal{X}_r$, $\mathcal{X}_f$ et $\mathcal{X}'_f$.
Intuition: une action modifie les variables flexibles. Les variables sans $'$ se réfèrent aux valeurs avant l'exécution de l'action, et les variables avec $'$ se réfèrent aux valeurs après l'exécution de l'action.
Cas spécial : une formule qui n'a aucune occurrence libre d'une variable de $\mathcal{X}'_f$ est appelée formule d'état.
Exemples: Soient $\mathcal{X}_r = \{m\}$ et $\mathcal{X}_f = \{x\}$.

- $x' = x + 1 \wedge m = x' - x$ est une formule d'action.
- $x = 5$ est une formule d'état.
- $\exists x'.x' = x + 1 \wedge m = x' - x$ est une formule d'état.

Sémantique

- On suppose une interprétation I des symboles de fonction et de prédicat ("obvious") et on note $|I|$ le domaine de valeurs (par exemple, les entiers mais aussi les fonctions comme l'addition).
- Un état $s : \mathcal{X}_f \rightarrow |I|$ est une valuation des variables flexibles. On en a besoin de deux, une "avant", s, une "après", $s' : \mathcal{X}'_f \rightarrow |I|$.
- On note par ξ une valuation des variables rigides.

Sémantique

- On suppose une interprétation I des symboles de fonction et de prédicat ("obvious") et on note $|I|$ le domaine de valeurs (par exemple, les entiers mais aussi les fonctions comme l'addition).
- Un état $s : \mathcal{X}_f \to |I|$ est une valuation des variables flexibles. On en a besoin de deux, une "avant", s, une "après", $s' : \mathcal{X}'_f \to |I|$.
- On note par ξ une valuation des variables rigides.
- Comme la vérité d'une formule A dépend de ces trois valuations, on la note par $[\![A]\!]^{\xi}_{s,s'}$.

Sémantique

- On suppose une interprétation I des symboles de fonction et de prédicat ("obvious") et on note $|I|$ le domaine de valeurs (par exemple, les entiers mais aussi les fonctions comme l'addition).
- Un état $s : \mathcal{X}_f \rightarrow |I|$ est une valuation des variables flexibles. On en a besoin de deux, une "avant", s, une "après", $s' : \mathcal{X}'_f \rightarrow |I|$.
- On note par ξ une valuation des variables rigides.
- Comme la vérité d'une formule A dépend de ces trois valuations, on la note par $[\![A]\!]^{\xi}_{s,s'}$.

Exemple: Soient $s = \{x \mapsto 4\}$, $s' = \{x' \mapsto 5\}$, $\xi = \{m \mapsto 1\}$. Alors

- $[\![x' = x + 1 \land m = x' - x]\!]^{\xi}_{s,s'} = 1$ (formule est vraie),
- $[\![x = 5]\!]^{\xi}_{s,s'} = 0$ (formule est fausse)
- $[\![\exists x'.x' = x + 1 \land m = x' - x]\!]^{\xi}_{s,s'} = 1$ (formule est vraie et ne dépend pas de s').

Rajouter des $'$...

Pour un terme ou formule e sans variables libres dans $\mathcal{X}'_f$, notons e' le terme/la formule obtenu(e) en rajoutant un $'$ à toutes les occurrences libres des variables dans $\mathcal{X}_f$.

Exemples (toutes les variables sont flexibles) :

$$\begin{array}{rcl}
(v+1)' & \equiv & v'+1 \\
(\exists x.n = x+m)' & \equiv & \exists x.n' = x+m' \\
(\exists n'.n = n'+m)' & \equiv & \exists np.n' = np+m'
\end{array}$$

Certaines formules importantes

Dans le model checking (voir TP), il y aura certaines formules qui ont une importance particulière.

Soit t un terme. On écrit:

- $[A]_t :\equiv A \vee t' = t$
- $\langle A \rangle_t :\equiv A \wedge \neg(t' = t)$

En particulier, t peut être une variable (flexible !) ou une liste de variables.

Certaines formules importantes

Dans le model checking (voir TP), il y aura certaines formules qui ont une importance particulière.

Soit t un terme. On écrit:

- $[A]_t :\equiv A \vee t' = t$
- $\langle A \rangle_t :\equiv A \wedge \neg(t' = t)$

En particulier, t peut être une variable (flexible !) ou une liste de variables.

Exemples:

- $[x' = x + 1]_x \equiv x' = x + 1 \vee x' = x$
- $\langle y' = x \cdot 2 \rangle_{x,y} \equiv y' = x \cdot 2 \wedge \neg(x' = x \wedge y' = y)$

Certaines formules importantes

Dans le model checking (voir TP), il y aura certaines formules qui ont une importance particulière.
Soit t un terme. On écrit:

- $[A]_t :\equiv A \vee t' = t$
- $\langle A \rangle_t :\equiv A \wedge \neg(t' = t)$

En particulier, t peut être une variable (flexible !) ou une liste de variables.
Exemples:

- $[x' = x + 1]_x \equiv x' = x + 1 \vee x' = x$
- $\langle y' = x \cdot 2 \rangle_{x,y} \equiv y' = x \cdot 2 \wedge \neg(x' = x \wedge y' = y)$

Notez que

$$\langle A \rangle_t \quad \equiv \quad \neg[\neg A]_t \qquad [A]_t \quad \equiv \quad \neg\langle \neg A \rangle_t$$

Exercise 13

Prouvez ces équivalences.

ENABLED

Pour une formule d'action A, on définit

$$\text{ENABLED } A \equiv \exists v'_1, \ldots, v'_n.A$$

où $v'_1, \ldots, v'_n$ sont toutes les variables libres avec $'$ qui apparaissent dans A.

Exercise 14

Est-ce que ENABLED A est "juste" une formule d'action ?

Notez que $\llbracket \text{ENABLED } A \rrbracket^{\xi}_{s,\_} = 1$ ssi il existe un état $s' : \mathcal{X}' \rightarrow |I|$ tel que $\llbracket A \rrbracket^{\xi}_{s,s'} = 1$.

Example

ENABLED $(x' = x + 1) \equiv \exists x'.x' = x + 1$. Cette formule est toujours vraie dans le domaine des entiers.
ENABLED $(x' = x + 1 \wedge x' = 5) \equiv \exists x'.x' = x + 1 \wedge x' = 5$. Cette formule est vraie $(\llbracket \exists x'.x' = x + 1 \wedge x' = 5 \rrbracket_{\overline{s,}\_} = 1)$ ssi $s(x) = 4$.
ENABLED $(x' = x + 1 \wedge x' = 5 \wedge x = 3) \equiv \exists x'.x' = x + 1 \wedge x' = 5 \wedge x = 3$.
Cette formule est insatisfiable (toujours fausse).

The Meaning of $\models$ (Again)

Recall that when you write $\ldots \models \varphi$, the idea is always the following:

- À gauche du $\models$, on fixe un cadre/monde/(ensemble de) modèle(s).
- La formule à droite du $\models$ est vraie dans ce monde $(\ldots)$.

The Meaning of $\models$ (Again)

Recall that when you write $\ldots \models \varphi$, the idea is always the following:

- À gauche du $\models$, on fixe un cadre/monde/(ensemble de) modèle(s).
- La formule à droite du $\models$ est vraie dans ce monde $(\ldots)$.

Maintenant, nous aurons à gauche une séquence d'états $\sigma = s_0 s_1 \ldots$.
Nous appelerons les formules temporelles simplement formules.

- Chaque formule d'état F est une formule.
  $\sigma, \xi \models F$ ssi $[\![F]\!]^{\xi}_{s_0,\_} = 1$
- Pour une formule d'action A et un terme t, $\Box[A]_t$ ("always square A sub t") est une formule.
  $\sigma, \xi \models \Box[A]_t$ ssi pour tout $n \in \mathbb{N}$, on a $[\![A]\!]^{\xi}_{s_n,s_{n+1}} = 1$ ou $[\![t]\!]^{\xi}_{s_n,\_} = [\![t]\!]^{\xi}_{s_{n+1},\_}$.
- Pour une formule d'état F, $\Box F$ est une formule.
  $\sigma, \xi \models \Box F$ ssi $[\![F]\!]^{\xi}_{s_n,\_} = 1$ pour tout $n \in \mathbb{N}$.

Notation (abréviations)

- $\Diamond F$ ("eventually F", "finally F") est une abréviation de $\neg\Box\neg F$.
- $\Diamond\langle A\rangle_t$ est une abréviation de $\neg\Box[\neg A]_t$
- $F \leadsto G$ ("F leads to G") est une abréviation de $\Box(F \longrightarrow \Diamond G)$.

Notez que $\Box$ correpond au **G** de LTL, et $\Diamond$ correspond au **F** de LTL.

Exemple de la sémantique d'une formule temporelle

Soit σ ainsi :

x	0	0	3	7	0	1	1	0	2	$\cdots$	(always $\neq 0$)
y	1	1	0	0	0	0	3	4	0	$\cdots$	(always $= 0$)

Lesquelles des formules suivantes sont vraies ?

- $\Box\neg(x = 0 \wedge y = 0)$
- $\Box[x = 0 \longrightarrow y' = 0]_{x,y}$
- $\Diamond(x = 7 \wedge y = 0)$
- $\Diamond\langle y = 0 \wedge x' = 0\rangle_y$
- $\Box\Diamond y \neq 0$
- $\Diamond\Box(x = 0 \longrightarrow y \neq 0)$
- $\Diamond\Box[\mathrm{false}]_y$

Fairness

- Une exécution est faiblement fair pour une action A si A a un nombre infini d'ocurrences pourvu que A finit par être toujours activée (enabled).
  Pour une action $\langle A\rangle_t$ (c'est-à-dire, une action A qui modifie t certainement) cela peut être écrit ainsi en TLA:
  $\Diamond\Box\text{ENABLED } \langle A\rangle_t \longrightarrow \Box\Diamond\langle A\rangle_t$.
- Une exécution est fortement fair pour une action A si A a un nombre infini d'ocurrences pourvu que A est activée (enabled) infiniment souvent.
  Pour une action $\langle A\rangle_t$ cela peut être écrit ainsi en TLA:
  $\Box\Diamond\text{ENABLED } \langle A\rangle_t \longrightarrow \Box\Diamond\langle A\rangle_t$.

TLA: Résumé

- TLA: la spécification de systèmes et les propriétés sont des formules.
- Formules d'action vs. formules temporelles.
- On peut définir la fairness.

Plan

1. Model Checking
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions
3. Abstract Interpretation and Deductive Verification

Plan

1. Model Checking
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions
3. Abstract Interpretation and Deductive Verification

Acknowledgements

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

Real-time (timed) models

Recall:

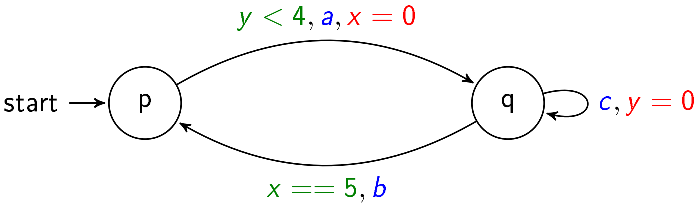
- Temporal models: we have a concept of time as a sequence of timepoints, i.e., the notion “X happens before Y”.
- Timed or real-time models: we have a quantitative concept of time, i.e., events happen “at a certain time”.

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

Exercise 15

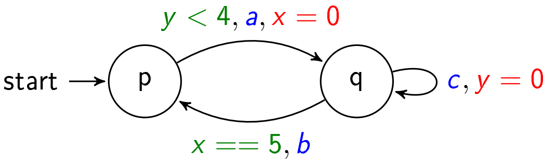
Where does the name “Uppaal” come from?

Example



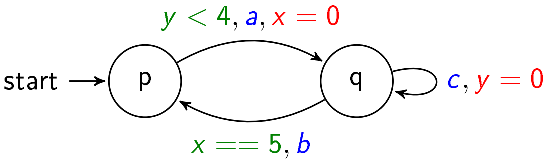
- locations (formerly: states): p, q.
- edges p → q, q → q, q → p.
  - edge p → q
    - guard:$y < 4$ *action* : a *reset* : $x = 0$
  - edge q → q
    - guard:**true** *action* : c *reset* : $y = 0$
  - edge q → p
    - guard:$x == 5$ *action* : b

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.

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

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

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 16

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

Exercise

Exercise 16

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

The definition on the next slide cannot be understood in a single pass. You will also have to look at the subsequent slides.

Timed automaton: Definition

Recall:

Definition

A Büchi automaton $\mathcal{A}$ is a tuple $(Q, q^o, \Sigma, \longrightarrow, F)$ in which:

- Q is a finite set of states;
- $q^o \in Q$ is the initial state;
- Σ is an alphabet;
- $\longrightarrow \subseteq \ Q \times \Sigma \times Q$ is the set of transitions;
- $F \subseteq Q$ is the set of acceptance states;

Timed automaton: Definition

Recall:

Definition

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

- Q is a finite set of states; We will now call them locations and hence use letter L;
- $q^o \in Q$ is the initial state; $\rightsquigarrow$ location ℓ^o;
- Σ is an alphabet of actions;
- X is the set of clock variables or simply clocks (see later . . .);
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions; we will now call them edges and they will have the form $L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ (see later . . .);
- $F \subseteq Q$ is the set of acceptance states; F will be trivial: $F = L$ and so we do not mention it;
- $\mathrm{Inv} : L \rightarrow \mathcal{C}(X)$ 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 17

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 18

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

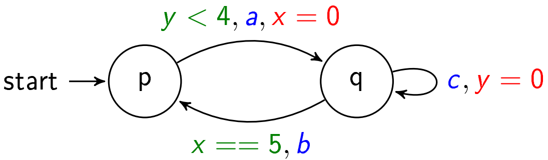
Constraint examples

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

Questions:

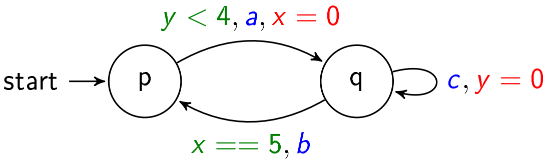
- Is $x \leq \pi$ a constraint?
- Can we write $x == 2$?
- Can we write $\neg(x == 2)$?

Timed automaton run (repeated)



Defining a state slightly more formally:

Timed automaton run (repeated)

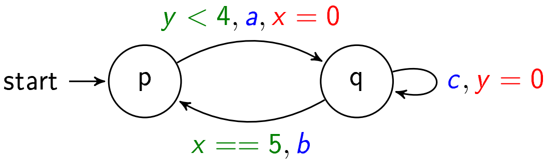


Defining a state slightly more formally:

A clock valuation is a function $\eta : X \to \mathbb{R}^+$.

A state is a pair $(\text{location}, \text{clock valuation})$.

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 $(\text{location}, \text{clock valuation})$.

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 19

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)

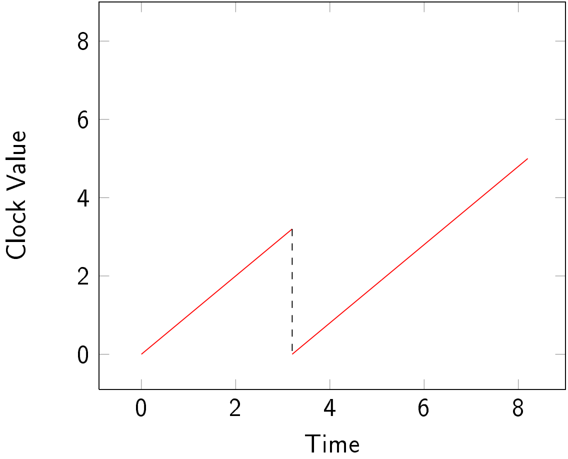
Given a run, e.g.

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

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

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

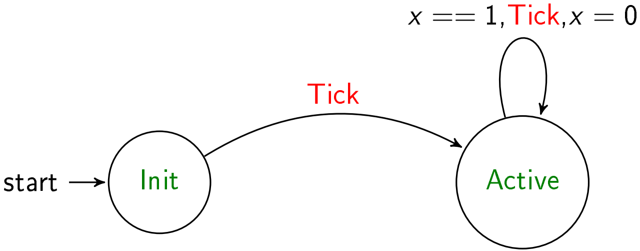


The clock evolves with time. It can be reset, but afterwards in continues to run immediately.

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:

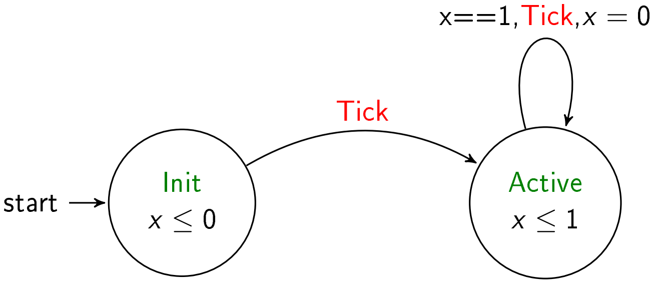


Exercise 20

Does this timed automaton enforce the behaviour described above?

Motivation for invariants (2)

S2:



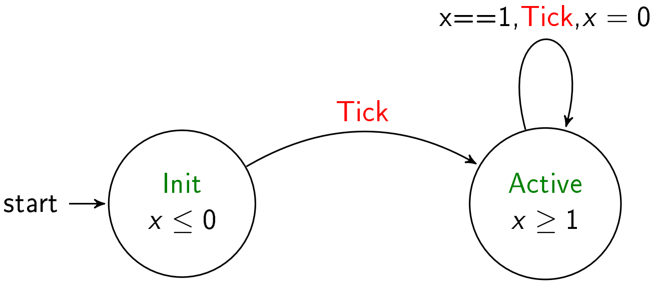
The invariant forces the automaton to stay in Active for at most one second before taking the edge going “Tick”.

Exercise 21

What are the traces/runs of this automaton?

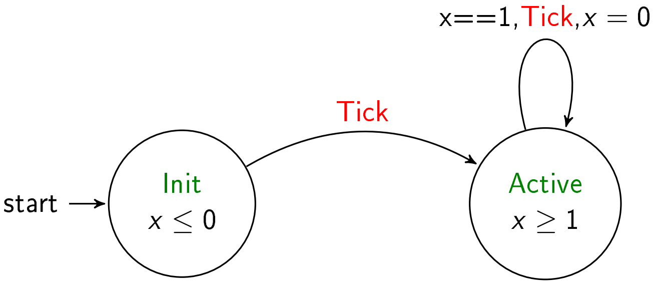
A strange invariant

S2:



A strange invariant

S2:



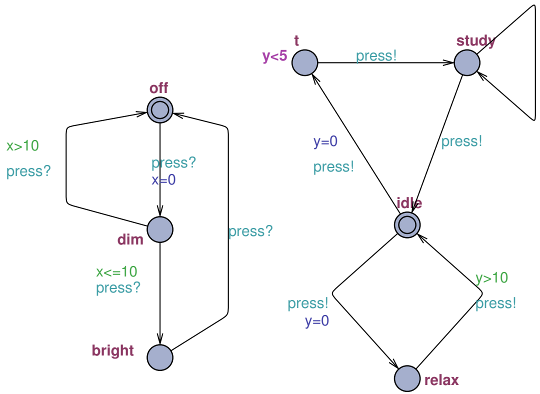
Following [BY04], we require that invariants have the form $x < c$ or $x \leq c$ (downwards-closed invariants).

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. Model Checking
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions
3. Abstract Interpretation and Deductive Verification

An example


off
x>10
press?
press?
x=0
dim
press?
x<=10
press?
bright
t
y<5
press!
study
y=0
press!
press!
idle
press!
y=0
y>10
press!
relax

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

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

Definition of product (composition)

We consider n timed automata $\mathcal{A}_i = (L_i, \ell_i^o, \Sigma, X, \longrightarrow_i, \mathrm{Inv}_i)$, $i = 1, \ldots, n$.
The product $\mathcal{A}_1 \| \ldots \| \mathcal{A}_n$ is the automaton $(L, \ell^o, \Sigma, X, \longrightarrow, \mathrm{Inv})$ 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 $((\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n), g_i \wedge g_j, c, r_i \cup r_j, (\ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n)) \in \longrightarrow$.
- $\mathrm{Inv}((\ell_1, \ldots, \ell_n)) = Inv_1(\ell_1) \wedge \ldots \wedge Inv_n(\ell_n)$;

Exercise 22

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

Plan

1. Model Checking
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions
3. Abstract Interpretation and Deductive Verification

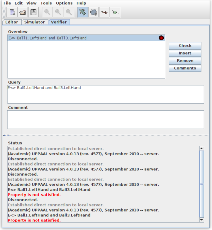
Uppaal

url: www.uppaal.com

Mature tool!

Graphical editor:

Verifier (model checker):

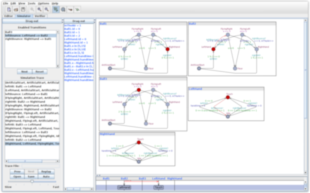


E<> Ball1.LeftHand and Ball3.LeftHand

Query: E<> Ball1.LeftHand and Ball3.LeftHand

```
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
E<> Ball1.LeftHand and Ball3.LeftHand
Property is not satisfied.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
E<> Ball1.LeftHand and Ball3.LeftHand
Property is not satisfied.
```

Simulator:



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 23

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.
The overall of this UE and in fact, the entire Master, is to verify that the actual behaviour corresponds to desired behaviour such as (recall):

- 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. The overall of this UE and in fact, the entire Master, is to verify that the actual behaviour corresponds to desired behaviour such as (recall):

- 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, because temporal logics have been treated before.

CTL subset

The language contains

- atomic state formulas:
  - $x \leq 3$, $y \leq 5$, $i == 10 \dots$
  - 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 24

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

Plan

1. Model Checking
2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions
3. Abstract Interpretation and Deductive Verification

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 $(\text{location}, \text{clock valuation}) \ldots$

Definition

Definition

Let $\mathcal{A} = (L, \ell^o, \Sigma, X, \longrightarrow, \mathrm{Inv})$. 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 "set")
- 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')
    - $\eta \models g$
    - $\eta' = [X' = 0]\eta$
    - $\eta' \models \mathrm{Inv}(\ell')$
  - delay transition: $(\ell, \eta) \xrightarrow{\delta} (\ell, \eta + \delta)$ with $\delta \in \mathbb{R}^+$ if $\forall d : 0 \leq d \leq \delta \Rightarrow \eta + d \models \mathrm{Inv}(\ell)$.

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 25

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.

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

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

Plan

1. Model Checking

2. Timed Automata
   - Basics
   - Composition of Timed Automata
   - Basics of the Tool Uppaal
   - Semantics
   - Case Study: LEGO Mindstorm
   - Regions

3. Abstract Interpretation and Deductive Verification

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 26

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 belt moves all the time at constant speed while the machine is switched on.
The arm makes a 360 degree rotation for a white brick. Of course, this rotation must be **timed** with reasonable precision so that the brick is actually kicked off.

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 27

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 28

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 29

(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.

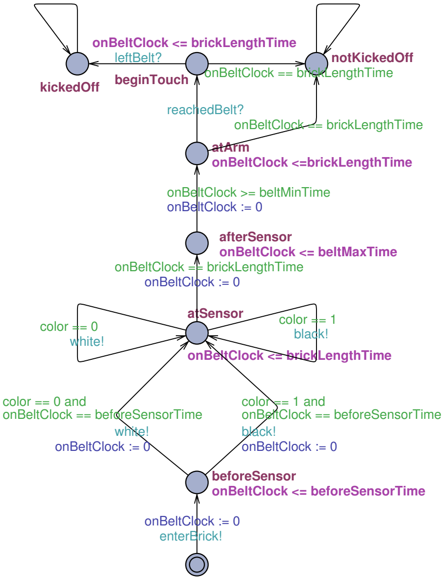
The Channels

- 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 30

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?

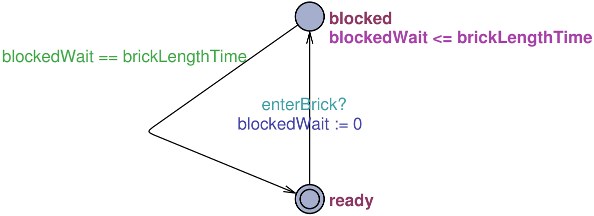
The Brick Template


onBeltClock <= brickLengthTime
leftBelt?
notKickedOff
kickedOff
beginTouch
onBeltClock == brickLengthTime
reachedBelt?
onBeltClock == brickLengthTime
atArm
onBeltClock <=brickLengthTime
onBeltClock >= beltMinTime
onBeltClock := 0
afterSensor
onBeltClock <= beltMaxTime
onBeltClock == brickLengthTime
onBeltClock := 0
atSensor
color == 0
white!
color == 1
black!
onBeltClock <= brickLengthTime
color == 0 and
onBeltClock == beforeSensorTime
white!
onBeltClock := 0
color == 1 and
onBeltClock == beforeSensorTime
black!
onBeltClock := 0
beforeSensor
onBeltClock <= beforeSensorTime
onBeltClock := 0
enterBrick!

Exercise 31

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

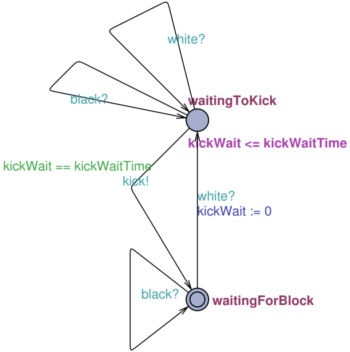
The Belt


blocked
blockedWait <= brickLengthTime
blockedWait == brickLengthTime
enterBrick?
blockedWait := 0
ready

Exercise 32

What is the belt process good for?

The Controller

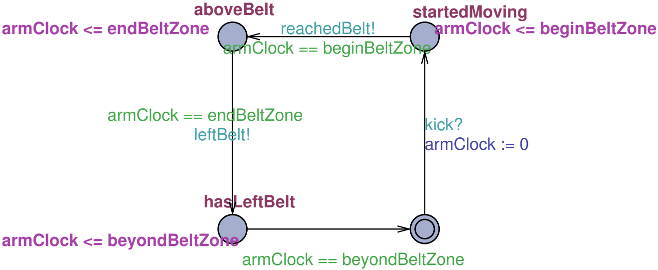


It could be envisaged that the translation of the program into the Uppaal process is done automatically as in [IKL+99].

Exercise 33

So how was the present controller model generated?

The Arm



Exercise 34

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 35

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.

Plan

1. Model Checking
2. Timed Automata
    - Basics
    - Composition of Timed Automata
    - Basics of the Tool Uppaal
    - Semantics
    - Case Study: LEGO Mindstorm
    - Regions
3. Abstract Interpretation and Deductive Verification

Region graph

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

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

Exercise 36

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 36

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 transititions.

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.

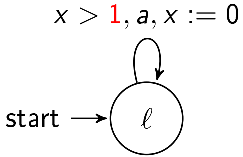
Thus, some assignments must be distinguished whereas others can be considered as equivalent, for our purposes.
Regions capture exactly this information.

Exercise 37

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

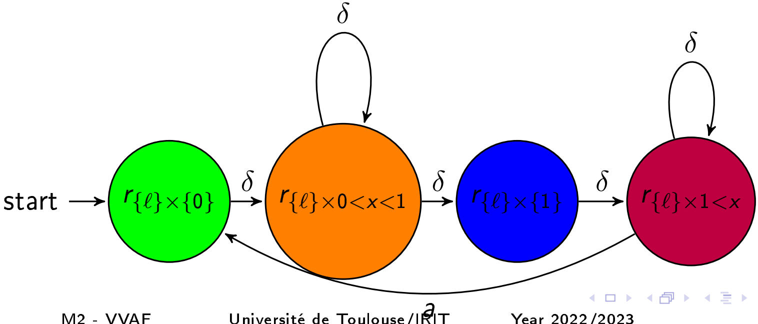
Consider the following automaton:



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:


start
$r_{\{\ell\}\times\{0\}}$
δ
$r_{\{\ell\}\times 0<x<1}$
δ
δ
$r_{\{\ell\}\times\{1\}}$
δ
$r_{\{\ell\}\times 1<x}$
δ
a

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 $\mathrm{fract}(t)$

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

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

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 ($\equiv_{\text{REG}}$)

Two valuations η and η' are region-equivalent: $\eta \equiv_{\text{REG}} \eta'$ 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 $\text{fract}(\eta(x)) = 0$ iff $\text{fract}(\eta'(x)) = 0$;
- for all x, y with $\eta(x) \leq C_x$ and $\eta(y) \leq C_y$, we have $\text{fract}(\eta(x)) \leq \text{fract}(\eta(y))$ iff $\text{fract}(\eta'(x)) \leq \text{fract}(\eta'(y))$.

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

Region equivalence relation

Visual illustration later . . .

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

Two valuations η and η' are region-equivalent: $\eta \equiv_{\text{REG}} \eta'$ 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 $\text{fract}(\eta(x)) = 0$ iff $\text{fract}(\eta'(x)) = 0$;
- for all x, y with $\eta(x) \leq C_x$ and $\eta(y) \leq C_y$, we have $\text{fract}(\eta(x)) \leq \text{fract}(\eta(y))$ iff $\text{fract}(\eta'(x)) \leq \text{fract}(\eta'(y))$.

Given a valuation η, the set of all valuations η' such that $\eta \equiv_{\text{REG}} \eta'$ 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 38

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

Determine which pairs of clock valuations are region-equivalent:

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

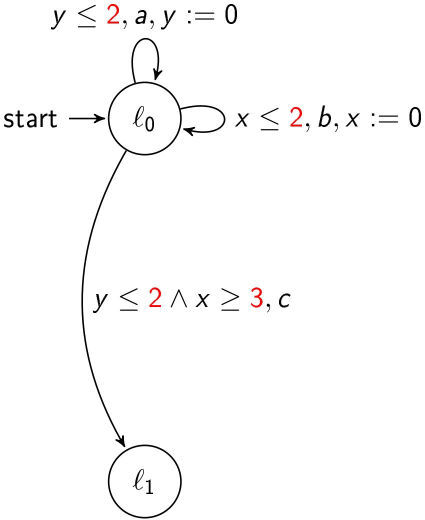
$3 : \{x \mapsto 1.3, y \mapsto 2.7, z \mapsto 5.5\}$ $4 : \{x \mapsto 1.4, y \mapsto 2.8, z \mapsto 4.5\}$

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

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

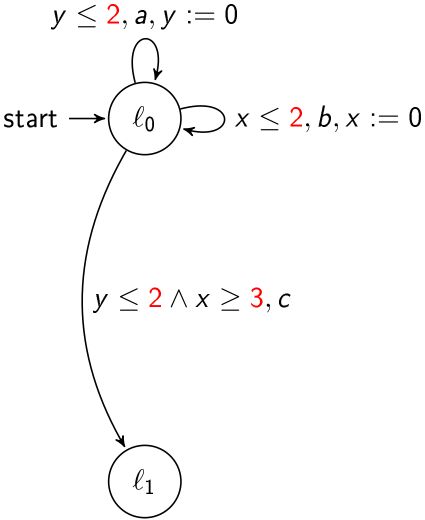
Region automaton example

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 $= \{x, y\}$.



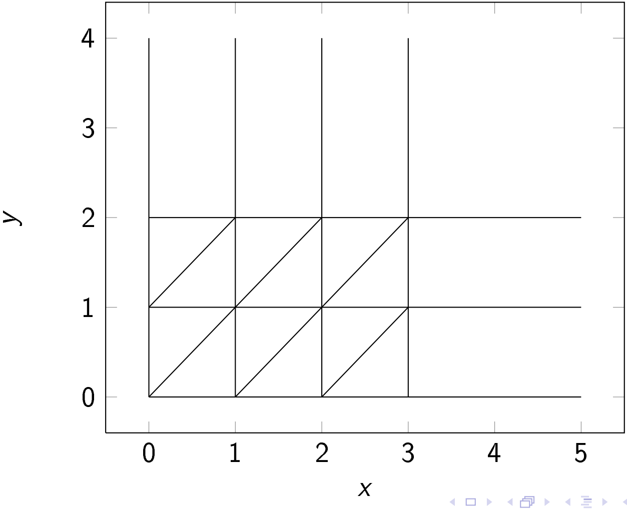
$$C_x = 3$$
$$C_y = 2$$

The regions of this example

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

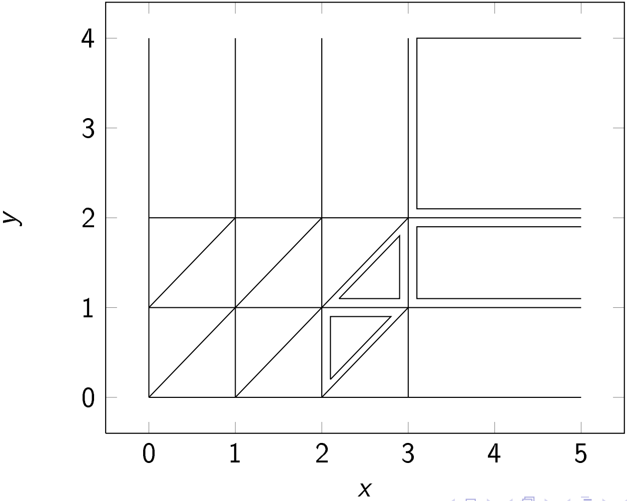
The regions of this example visualised

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



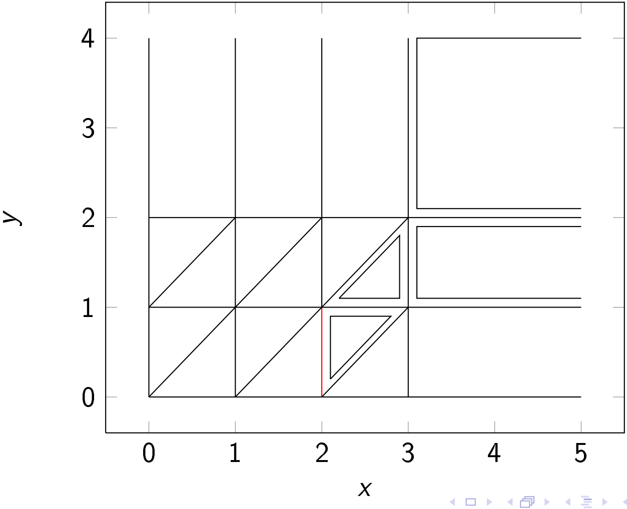
Time successors in this example

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:



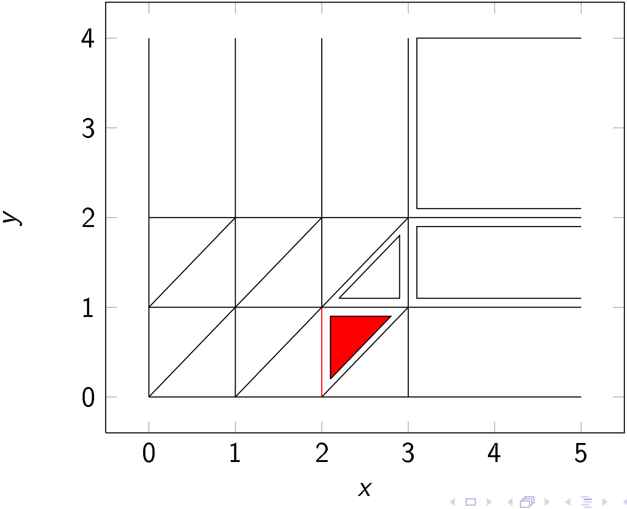
Time successors in this example

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:



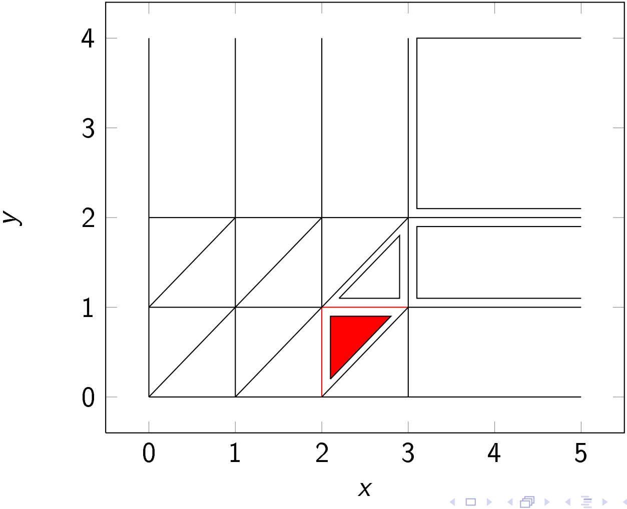
Time successors in this example

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:



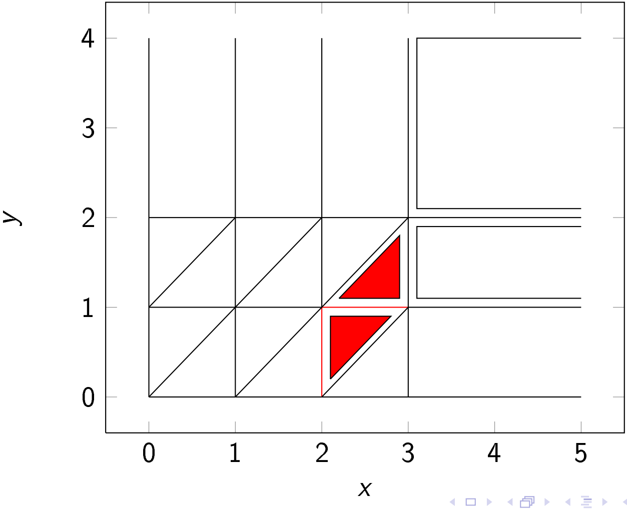
Time successors in this example

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:



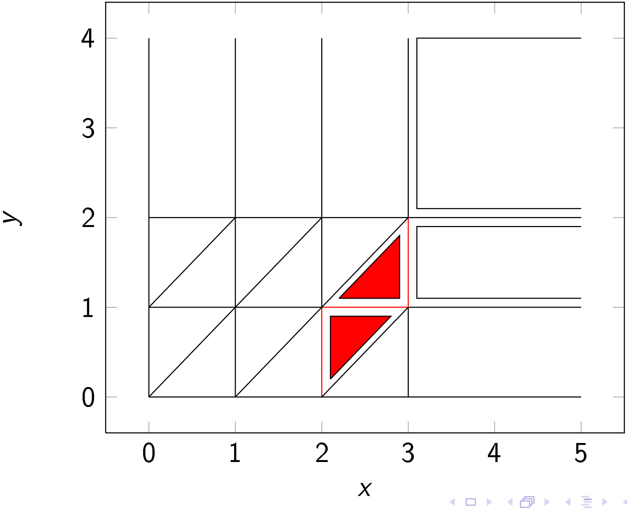
Time successors in this example

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:



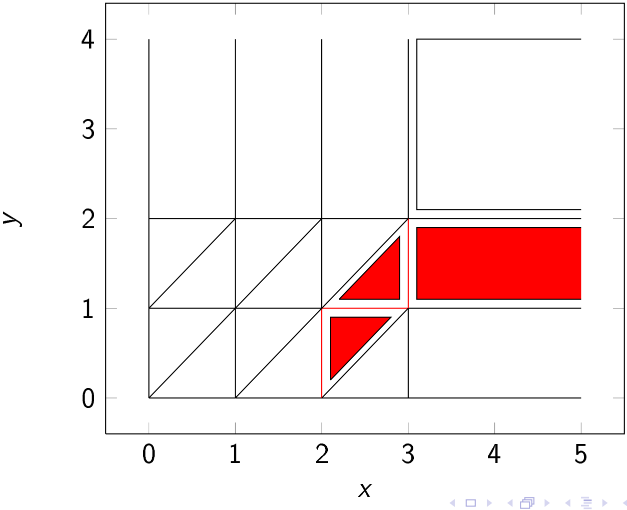
Time successors in this example

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:



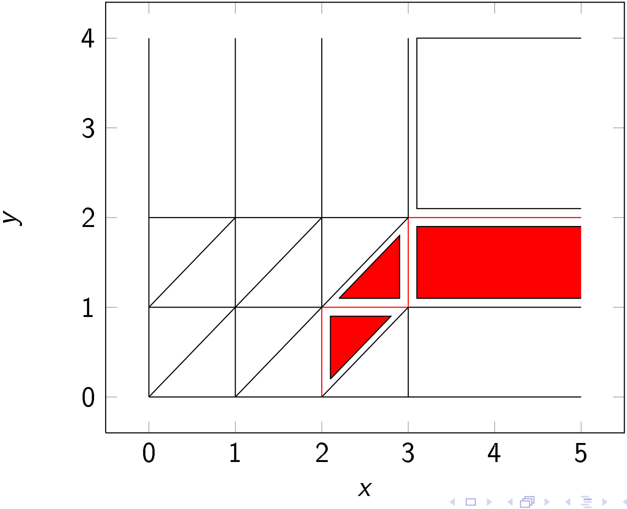
Time successors in this example

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:



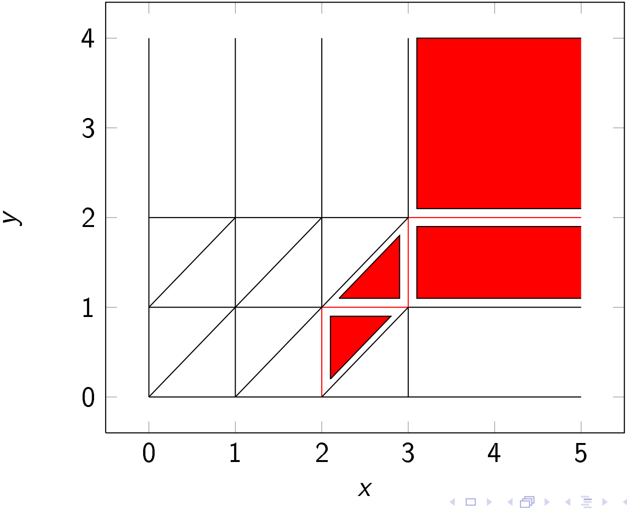
Time successors in this example

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 39

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.

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. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

Acknowledgements

The slides and exercises of this chapter are based on material by Loïc Correnson, Nikolai Kosmatov, A. Miné, Allan Blanchard [Bla19] and Pierre Roux, and on [KKP$^+$15, Lei05].

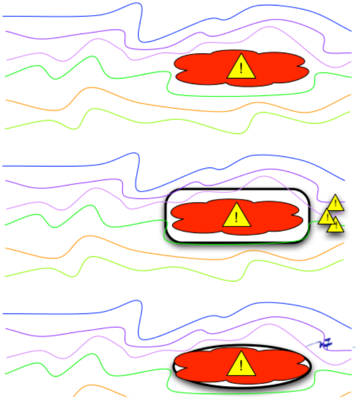
Plan

1. Model Checking

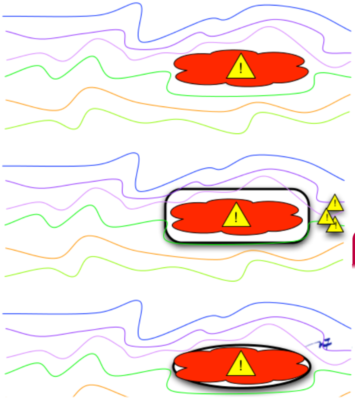
2. Timed Automata

3. Abstract Interpretation and Deductive Verification
    - Abstract Interpretation: Data Abstraction I
    - Abstract Interpretation: Program Abstraction
    - Abstract Interpretation: Data Abstraction II
    - Abstract Interpretation: Executing an Abstract Program
    - Frama-C
    - Frama-C and EVA
    - Deductive Verification
    - Frama-C and Deductive Verification

Abstract Interpretation at a glance



Abstract Interpretation at a glance



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 40

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?

Another Illustration

$$
\begin{array}{l}
{}_{0}x = \textbf{rand}(0,\ 12)\ ;{}_{1}y = 42\ ; \\
\textbf{while}\ {}_{2}(x > 0)\ \{ \\
\qquad {}_{3}x = x - 2\ ; \\
\qquad {}_{4}y = y + 4\ ; \\
\}_{5}
\end{array}
$$

Another Illustration

$$
\begin{array}{l}
{}_{0}\mathsf{x} = \mathbf{rand}(0,\ 12)\,;{}_{1}\mathsf{y} = 42\,; \\
\mathbf{while}\ {}_{2}(\mathsf{x} > 0)\ \{ \\
\qquad {}_{3}\mathsf{x} = \mathsf{x} - 2\,; \\
\qquad {}_{4}\mathsf{y} = \mathsf{y} + 4\,; \\
\}_{5}
\end{array}
$$

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

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 (PNZ-domain): `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 41

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 42

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 42

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 43

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

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 44

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 45

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

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: $\alpha(\{1,7\}) = \mathtt{P}$, $\alpha(\{1,7,9,10\}) = \mathtt{P}$, $\ldots$

Exercise 46

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

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 $\alpha(\{1,7\}) = \mathtt{P}$, $\alpha(\{1,7,9,10\}) = \mathtt{P}$, $\alpha(\{1,7,9,10,11,25\}) = \mathtt{P}$, $\ldots$, 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 47

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

Order $\sqsubseteq^\sharp$ between abstract objects

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

Definition ($\sqsubseteq^\sharp$)

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

Order $\sqsubseteq^\sharp$ between abstract objects

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

Definition ($\sqsubseteq^\sharp$)

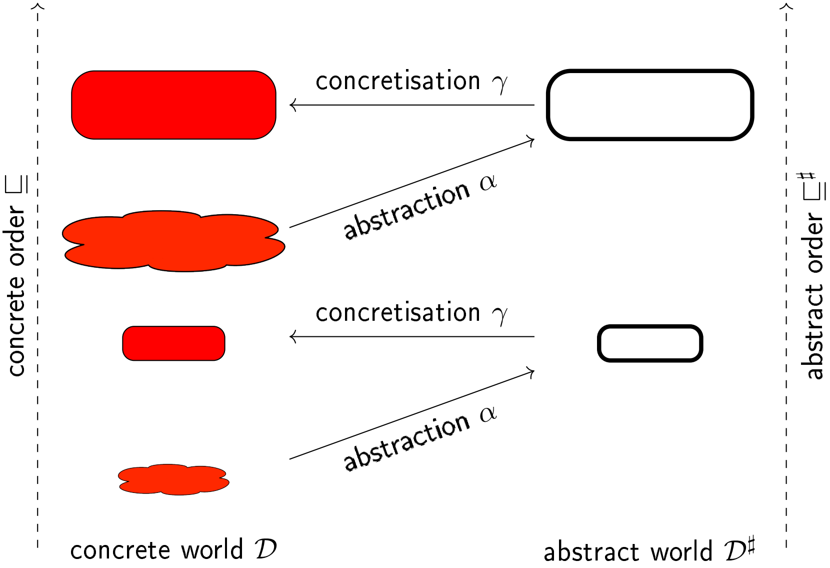
$$\forall o^\sharp, o^{\sharp\prime}, \quad o^\sharp \sqsubseteq^\sharp o^{\sharp\prime} :\Leftrightarrow \gamma(o^\sharp) \sqsubseteq \gamma(o^{\sharp\prime})$$

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

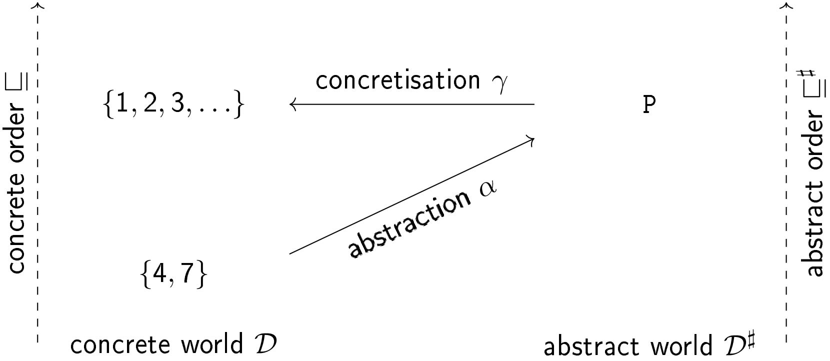
Exercise 48

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

Concrete — abstract : summary


concretisation γ
abstraction α
concretisation γ
abstraction α
concrete order $\sqsubseteq$
abstract order $\sqsubseteq^\sharp$
concrete world $\mathcal{D}$
abstract world $\mathcal{D}^\sharp$

Concrete — abstract : summary on PNZ-example



concrete order $\sqsubseteq$

$\{1, 2, 3, \ldots\}$

concretisation γ

P

abstraction α

$\{4, 7\}$

abstract order $\sqsubseteq^\sharp$

concrete world $\mathcal{D}$

abstract world $\mathcal{D}^\sharp$

Abstract operations

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

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

We will get back to this later ...

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

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

$n \in \mathbb{Z}$

$\textbf{rand}(n_1,\ n_2)$ simulates an input value.

Exercise 49

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

Toy language example

Example

```
x = rand(0, 12); y = 42;
while (x > 0) {
      x = x - 2;
      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 $\mathbb{V} \to \mathbb{Z}$.

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 $\mathbb{Z}$.

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 $\mathbb{Z}$. So here is the signature of the semantics of expressions : $[\![e]\!]_{\mathrm{E}} : (\mathbb{V} \rightarrow \mathbb{Z}) \rightarrow 2^{\mathbb{Z}}$

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

...

Exercise 50

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:

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

Type of the concrete program semantics

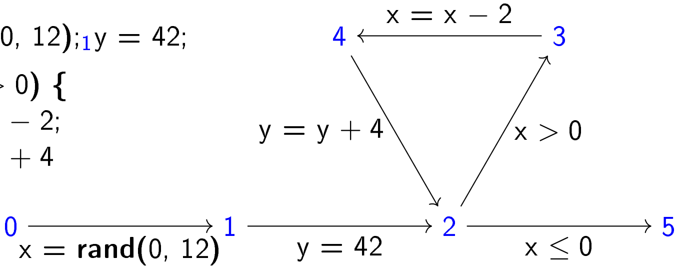
The concrete semantics is of this type:

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

- 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

```
$_0$x = **rand**(0, 12);$_1$y = 42;
**while** $_2$(x > 0) {
    $_3$x = x − 2;
    $_4$y = y + 4
}$_5$
```



Denoting by S_i the semantics at point i:

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

Exercise 51

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

  ⇒ 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 52

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

Two solutions:

- Abstract $2^{\mathbb{V}\to\mathbb{Z}}$ directly into one $\mathcal{D}^\sharp$
  - relational : certain combinations of x and y are impossible
  - + more precise
  - − more complicated and costly

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

Two solutions:

- Abstract $2^{\mathbb{V}\to\mathbb{Z}}$ directly into one $\mathcal{D}^\sharp$
  - relational : certain combinations of x and y are impossible

  $+$ more precise

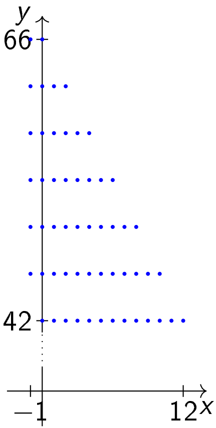
  $-$ more complicated and costly
- Abstract $2^{\mathbb{V}\to\mathbb{Z}}$ into $\mathbb{V} \to 2^{\mathbb{Z}}$, and then $2^{\mathbb{Z}}$ into one $\mathcal{D}^\sharp$
  - non-relational : the values of x and y are independent.
  - Current state of affairs in Frama-C (leaving aside some special settings working for toy examples).

Exercise 53

Discuss this issue using the two forum exercises above.

Two pictures are worth a thousand words

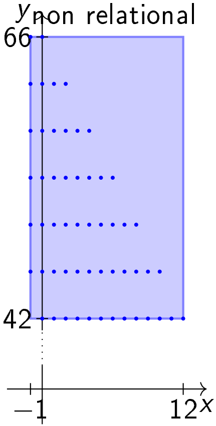
Previous example at program point 2 (loop invariant)



Two pictures are worth a thousand words

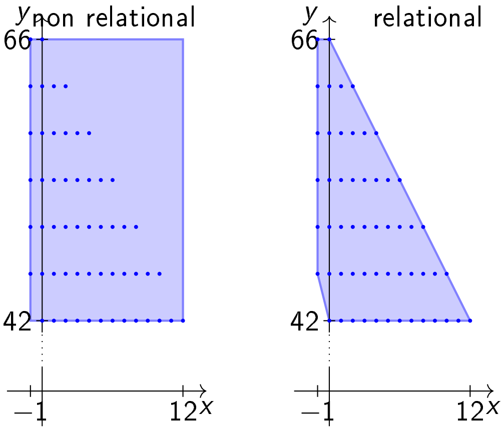
Previous example at program point 2 (loop invariant)

non relational



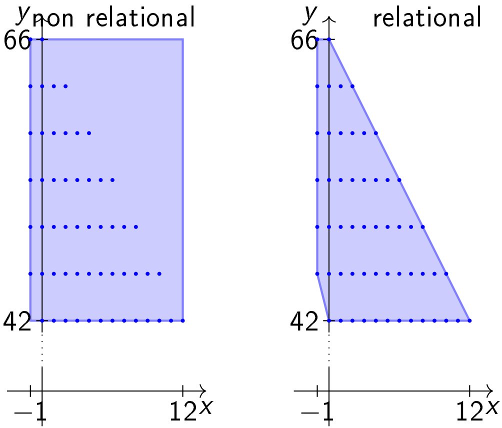
Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



Two pictures are worth a thousand words

Previous example at program point 2 (loop invariant)



Exercise 54

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

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

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

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

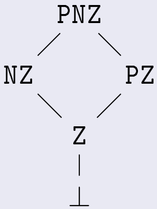
- We have looked at some basic ideas of data abstraction.
- We have looked at program abstraction.
- We will now get back to data abstraction, to understand how abstract domains are set up and give a couple of examples of abstract domains.
- The presentation above suggests that concrete data comes first, 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 (treillis) for $\mathcal{D}^\sharp = \{\mathtt{PNZ}, \mathtt{NZ}, \mathtt{PZ}, \mathtt{Z}, \bot\}$:



PNZ

NZ

PZ

Z

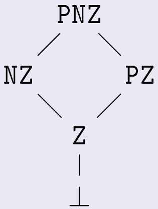
$\bot$

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

Domain of signs

Definition

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



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

The definition of α follows from the definition of γ as stated above:

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

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

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

$+^\sharp$	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 55

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

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	$\perp$
PNZ					
NZ					
PZ					
Z					
$\perp$					

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:

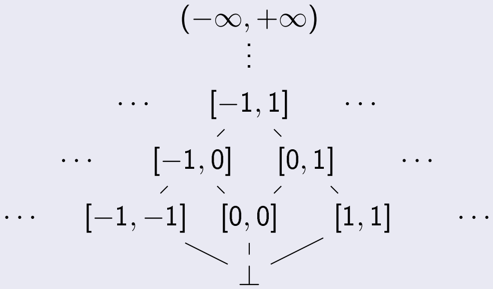
$/^\sharp$	PNZ	NZ	PZ	Z	⊥
PNZ	PNZ	PNZ	PNZ	⊥	⊥
NZ	PNZ	PZ	NZ	⊥	⊥
PZ	PNZ	NZ	PZ	⊥	⊥
Z	Z	Z	Z	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥

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



$(-\infty, +\infty)$

$\vdots$

$\cdots \quad [-1, 1] \quad \cdots$

$\cdots \quad [-1, 0] \quad [0, 1] \quad \cdots$

$\cdots \quad [-1, -1] \quad [0, 0] \quad [1, 1] \quad \cdots$

$\bot$

$$\begin{aligned}
\gamma((-\infty, +\infty)) &= (-\infty, +\infty) & \gamma([n_1, n_2]) &= [n_1, n_2] \\
\gamma((-\infty, n]) &= (-\infty, n] & \gamma(\bot) &= \emptyset \\
\gamma([n, +\infty)) &= [n, +\infty)
\end{aligned}$$

Domain of intervals: abstraction

We define:

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

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

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

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

- **Exercise 56**

  Subtraction, multiplication

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

Reminder: The Toy Program

```
$_0x = \mathbf{rand}(0, 12);_1y = 42;$
while $_2(x > 0)$ {
    $_3x = x - 2;$
    $_4y = y + 4$
}$_5$
```

Executing the Program Using the Interval Domain

	x	y		x	y		x	y
0:	$(-\infty,\infty)$	$(-\infty,\infty)$						
1:	$[0,12]$	$(-\infty,\infty)$						
2^*:	$[0,12]$	$[42,42]$	2^*:	$[-1,8]$	$[50,50]$	2^*:	$[-1,4]$	$[58,58]$
3:	$[1,12]$	$[42,42]$	3:	$[1,8]$	$[50,50]$	3:	$[1,4]$	$[58,58]$
4:	$[-1,10]$	$[42,42]$	4:	$[-1,6]$	$[50,50]$	4:	$[-1,2]$	$[58,58]$
2^*:	$[-1,10]$	$[46,46]$	2^*:	$[-1,6]$	$[54,54]$	2^*:	$[-1,2]$	$[62,62]$
3:	$[1,10]$	$[46,46]$	3:	$[1,6]$	$[54,54]$	3:	$[1,2]$	$[62,62]$
4:	$[-1,8]$	$[46,46]$	4:	$[-1,4]$	$[54,54]$	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 $\boxed{5\text{: } [0,0]\ [42,42]}$, $\boxed{5\text{: } [-1,0]\ [46,46]}$, $\boxed{5\text{: } [-1,0]\ [50,50]}$, $\boxed{5\text{: } [-1,0]\ [54,54]}$, $\boxed{5\text{: } [-1,0]\ [58,58]}$, $\boxed{5\text{: } [-1,0]\ [62,62]}$, so in summary $\boxed{5\text{: } [-1,0]\ [42,66]}$.

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 179).
- 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. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

Confinement Exercises

In the sequel there are very few forum or mandatory exercises, as the introduction to a software tool does not really lend itself to such exercises.
It would be like learning to swim without water!
It is during the TP sessions that you will have exercises!

Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2012: WP: Weakest-precondition based plugin
- 2012: E-ACSL: Runtime Verification plugin
- 2013: CEA Spin-off TrustInSoft
- 2016: Eva: Evolved Value Analysis
- 2016: Frama-Clang: C++ extension
- Today: Frama-C Titanium (version 22.0, 2020)

Frama-C, a Collection of Tools

Several tools inside a single platform

- tools provided as plug-ins
  - 21 plug-ins in the open source distribution
  - outside open source plug-ins (E-ACSL & Frama-Clang, a few others)
  - 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 and wp for deductive verification.

ACSL: Introduction

- ACSL = ANSI/ISO C Specification Language
- Inspired by Java's JML and Eiffel function contracts.
- First-order logic.

ACSL: Introduction

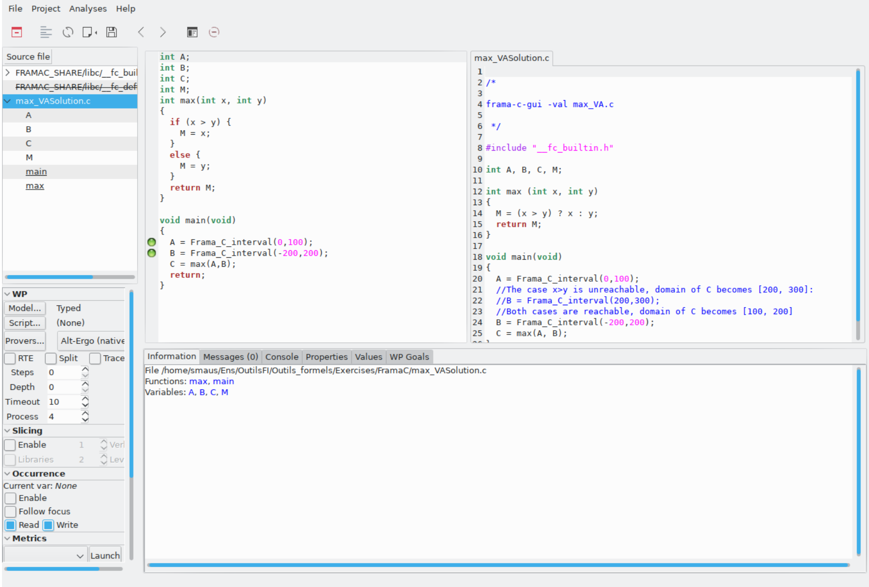
- ACSL = ANSI/ISO C Specification Language
- Inspired by Java's JML and Eiffel function contracts.
- First-order logic.
- Specification of a function states pre-conditions and post-conditions.
- Special clause for stating which memory locations the function assigns (i.e. any memory location not mentioned must be unmodified).

ACSL: Simple example

```
/*@ requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *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
- `assigns` : all other memory locations must be unaltered
- `ensures` and `\at`, `Pre` (entry point of function)

Screenshot



Model... Typed
Script... (None)
Provers... Alt-Ergo (native
□ RTE □ Split □ Trace
Steps 0
Depth 0
Timeout 10
Process 4

Slicing
□ Enable 1 Verl
□ Libraries 2 Lev

Occurrence
Current var: *None*
□ Enable
□ Follow focus
☑ Read ☑ Write

```
int A;
int B;
int C;
int M;
int max(int x, int y)
{
  if (x > y) {
    M = x;
  }
  else {
    M = y;
  }
  return M;
}

void main(void)
{
  A = Frama_C_interval(0,100);
  B = Frama_C_interval(-200,200);
  C = max(A,B);
  return;
}
```

```

/*

frama-c-gui -val max_VA.c

 */

#include "__fc_builtin.h"

int A, B, C, M;

int max (int x, int y)
{
  M = (x > y) ? x : y;
  return M;
}

void main(void)
{
  A = Frama_C_interval(0,100);
  //The case x>y is unreachable, domain of C becomes [200, 300]:
  //B = Frama_C_interval(200,300);
  //Both cases are reachable, domain of C becomes [100, 200]
  B = Frama_C_interval(-200,200);
  C = max(A, B);
```

File /home/smaus/Ens/OutilsFI/Outils_formels/Exercises/FramaC/max_VASolution.c
Functions: max, main
Variables: A, B, C, M

ACSL: Behaviors

```
/*@ behavior not_null:
    assumes a!=\null && b!= \null;
    requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *a, *b;
    ensures *a == \at(*b,Pre) && *b == \at(*a,Pre);

    behavior null:
    assumes a == \null || b == \null;
    assigns \nothing;

    complete behaviors;
    disjoint behaviors;
*/
void swap_or_null(int * a, int * b);
```

ACSL: Behaviors (2)

- `behaviors` are a structuring mechanism. They make specifications more readable (see TP).
- Can be guarded by `assumes` clauses.
- Behaviors need not be `complete` or `disjoint`, but this can be stipulated.

ACSL: Loop invariants

```
int a[10];
/*@
  loop invariant 0 <= i <= 10;
  loop invariant \forall integer j; 0 <= j < i ==> a[j] == j;
  loop assigns i, a[0 .. i-1];
*/
for (int i = 0; i < 10; i++) a[i] = i;
```

- `loop invariant`s 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.
- `loop assigns` is special case of invariant stating which memory locations might be modified by the loop.
- Particularly useful for deductive verification.

Communication between plug-ins

Each plug-in can provide a validity status to any ACSL property and/or generate ACSL annotations. This lets ACSL annotations play an important role in the communication between plug-ins (in our case: EVA for abstract interpretation and wp for deductive verification).

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.
An `assert` is sometimes inserted by Frama-C, but we can also insert it manually.

Meaning of circles

- Blue circle: no verification has been attempted
- Green bullet: specification is ensured by the program (but wp analysis does not guarantee absence of runtime errors)
- Green+brown bullet: proof has been produced but it can depend on some properties that have not been verified.
- Orange bullet: no prover could determine if the property is verified. There are three possibles reasons:
    - the prover did not have enough information,
    - the prover did not have enough time to compute the proof and encountered a timeout
    - the property does not hold.
- Green+blue bullet: it is an axiom; considered as being true.

Why does it not work?

Blanchard [Bla19]: When automatic solvers cannot ensure that our properties are verified, it it sometimes hard to understand why. Indeed, provers are generally not able to answer something other than “yes”, “no” or “unknown”, they are not able to extract the reason of a “no” or an “unknown”. There exist tools that can explore a proof tree to extract this type of information, currently Frama-C does not provide such a tool.

Resources

The following can happen: you add an assert A which Frama-C is able to prove. Then you add an assertion B and suddenly, Frama-C in not able to prove A anymore. This non-monotonicity seems strange at first.
It is a problem of resources: Frama-C devotes a specified time to proving the assertions, and in the presence of B, it may be that A does not get enough time.
Can be controlled with `-wp-timeout`.

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
    - Abstract Interpretation: Data Abstraction I
    - Abstract Interpretation: Program Abstraction
    - Abstract Interpretation: Data Abstraction II
    - Abstract Interpretation: Executing an Abstract Program
    - Frama-C
    - Frama-C and EVA
    - Deductive Verification
    - Frama-C and Deductive Verification

The EVA plug-in

- Call frama-c (with our without GUI) with option `-val` for value analysis.

The EVA plug-in

- Call frama-c (with our without GUI) with option `-val` for value analysis.
- Purposes:
  - get familiar with foreign code;
  - produce documentation automatically (not treated here);
  - search for bugs;
  - guarantee the absence of bugs.

Abstract execution: Information loss

Compared to the theory of abstract interpretation, Frama-C is much more pragmatic. Users cannot define their own domains (might change in the future), but some pre-defined domains are used.

Abstract execution: Information loss

Compared to the theory of abstract interpretation, Frama-C is much more pragmatic. Users cannot define their own domains (might change in the future), but some pre-defined domains are used.
A simple case to begin with is the abstract domain $2^{\mathbb{Z}}$. The program semantics $2^{\mathbb{V}\to\mathbb{Z}}$ is abstracted into $\mathbb{V} \to 2^{\mathbb{Z}}$, i.e., for each program point the set of possible values is computed but the variables are independent (non-relational domain).
Hence if $x \in \{1, 3\}$ and $y \in \{1, 3\}$ then Frama-C will compute that $x + y \in \{2, 4, 6\}$ (of course), but also $x + x \in \{2, 4, 6\}$.

Abstract execution: Information loss

Compared to the theory of abstract interpretation, Frama-C is much more pragmatic. Users cannot define their own domains (might change in the future), but some pre-defined domains are used.
A simple case to begin with is the abstract domain $2^{\mathbb{Z}}$. The program semantics $2^{\mathbb{V}\to\mathbb{Z}}$ is abstracted into $\mathbb{V} \to 2^{\mathbb{Z}}$, i.e., for each program point the set of possible values is computed but the variables are independent (non-relational domain).
Hence if $x \in \{1,3\}$ and $y \in \{1,3\}$ then Frama-C will compute that $x + y \in \{2,4,6\}$ (of course), but also $x + x \in \{2,4,6\}$.
So far, loss of information for two reasons:

- Set of values for each program point because the point may be passed more than once;
- non-relational domain.

Abstract Domains

And yet, the abstract domain $2^{\mathbb{Z}}$ is a complex one: for $x \in \{1, 3, 5, 7\}$ we have $x \times x = \{1, 3, 5, 7, 9, 15, 21, 25, 35, 49\}$. No “data abstraction” has taken place yet!

Abstract Domains

And yet, the abstract domain $2^{\mathbb{Z}}$ is a complex one: for $x \in \{1, 3, 5, 7\}$ we have $x \times x = \{1, 3, 5, 7, 9, 15, 21, 25, 35, 49\}$. No "data abstraction" has taken place yet!

By default, Frama-C represents integer sets up to size 8. One may increase the size or represented sets using the option `-val-ilevel`.

Abstract Domains

And yet, the abstract domain $2^{\mathbb{Z}}$ is a complex one: for $x \in \{1, 3, 5, 7\}$ we have $x \times x = \{1, 3, 5, 7, 9, 15, 21, 25, 35, 49\}$. No "data abstraction" has taken place yet!
By default, Frama-C represents integer sets up to size 8. One may increase the size or represented sets using the option `-val-ilevel`.
Any bigger set is approximated by an interval with periodicity information:

$$[l..u], r\%m$$

is the set of values comprised between l and u that are congruent to r modulo m. May seem an ad-hoc choice but may be quite useful in practice in particular when talking about memory addresses and arrays.

slevel and ulevel

- The `-ulevel n` option (syntactic unrolling) unrolls any loop in the analysed program `n` times, which is actually visible in the Frama-C GUI middle window containing the pre-processed version of the source code.

slevel and ulevel

- The `-ulevel n` option (syntactic unrolling) unrolls any loop in the analysed program `n` times, which is actually visible in the Frama-C GUI middle window containing the pre-processed version of the source code.
- The `-slevel n` option (semantic unrolling) is more general and means that for any program point, Frama-C may separate `n` states. In particular, the two branches of an if-then-else may be separated. This increases the precision and may boil down to having a relational domain although this is technically not the case. E.g. if the then-branch results in $x = 0, y = 10$ and the else-branch results in $x = 10, y = 0$, then an slevel of 2 will result in $x + y = 10$, rather than just $x + y \in \{0, 10, 20\}$.
  `slevel` is an option of the EVA plugin. In practice, a level of 100 is no problem.

slevel and ulevel

- The `-ulevel n` option (syntactic unrolling) unrolls any loop in the analysed program `n` times, which is actually visible in the Frama-C GUI middle window containing the pre-processed version of the source code.
- The `-slevel n` option (semantic unrolling) is more general and means that for any program point, Frama-C may separate `n` states. In particular, the two branches of an if-then-else may be separated. This increases the precision and may boil down to having a relational domain although this is technically not the case. E.g. if the then-branch results in $x = 0, y = 10$ and the else-branch results in $x = 10, y = 0$, then an slevel of 2 will result in $x + y = 10$, rather than just $x + y \in \{0, 10, 20\}$.
  `slevel` is an option of the EVA plugin. In practice, a level of 100 is no problem.

In practice the effects of slevel and ulevel may be similar, see TP.

Abstract Interpretation and Memory Access

Errors due to illegal memory access (e.g. using a too high index for an array) are quite common, yet often relatively easy to detect, as the math involved in calculating indices is usually not highly complex.
It gives a use case par excellence for abstraction interpretation.
See TP.

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

Hoare triples

Proposed in 1969. Expressions of the form $\{P\}C\{Q\}$, where P and Q are logic formulas that express properties about the memory at particular program points, C. States: P is a sufficient precondition to ensure that C will bring us to the postcondition Q (termination provided).

Hoare triples

Proposed in 1969. Expressions of the form $\{P\}C\{Q\}$, where P and Q are logic formulas that express properties about the memory at particular program points, C. States: P is a sufficient precondition to ensure that C will bring us to the postcondition Q (termination provided).
How to mechanize?

Weakest precondition calculus

Developed by Dijkstra. Tells us how to transform Q into P, given C (backward reasoning). The weakest precondition for obtaining Q is computed.

Some examples (syntax of our toy language):

stm	$wp(stm, Q)$
$x = E$	$Q[x := E]$
$S \; T$	$wp(S, wp(T, Q))$
if $(expr > 0)$ **{** S **}** **else** **{** T **}**	$expr > 0 \longrightarrow wp(S, Q) \wedge$ $expr \leq 0 \longrightarrow wp(T, Q)$
...	...

Here $Q[x := E]$ denotes substitution of x by E in Q.

Explaining *wp* for an assignment

One might be confused by the fact that the weakest precondition for having Q after an assignment $x = E$ is $Q[x := E]$.
In [Lei05] this is explained like this:

> $Q[x := E]$ *says about* E *what* Q *says about* x.

Explaining *wp* for an assignment

One might be confused by the fact that the weakest precondition for having Q after an assignment $x = E$ is $Q[x := E]$.
In [Lei05] this is explained like this:

> $Q[x := E]$ *says about* E *what* Q *says about* x.

Example: $Q :\equiv (x == 2 \wedge y == 2)$ and $E :\equiv y$. The weakest precondition for having Q after the assignment $x = y$ is indeed
$Q[x := E] \equiv (y == 2 \wedge y == 2) \equiv (y == 2)$.

Plan

1. Model Checking
2. Timed Automata
3. Abstract Interpretation and Deductive Verification
   - Abstract Interpretation: Data Abstraction I
   - Abstract Interpretation: Program Abstraction
   - Abstract Interpretation: Data Abstraction II
   - Abstract Interpretation: Executing an Abstract Program
   - Frama-C
   - Frama-C and EVA
   - Deductive Verification
   - Frama-C and Deductive Verification

WP plugin

- Hoare-logic based plugin, developed at CEA List
- Proof of semantic properties of the program
- Modular verification (function by function)
- Input: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on automatic theorem provers to discharge the VCs: Alt-Ergo, Simplify, Z3, Yices, CVC3, CVC4 . . .
- WP manual at `http://frama-c.com/wp.html`
- If all VCs are proved, the program respects the given specification.

Finding the right preconditions

Precondition with `requires`.
Postcondition with `ensures`.

Finding the right preconditions

Precondition with `requires`.
Postcondition with `ensures`.
Finding the right preconditions for a function is sometimes hard.
Danger: building a specification that would contain the same bugs currently existing in the source code.
Good practice: working with another person, one for the code and one for the specification.

Semantic Correctness

Showing that a program computes what it is supposed to:

```
/*@
  ensures ( x > y ==> \result == x );
  ensures ( x <=y ==> \result == y );
*/
int maxint (int x, int y) {
  return (x > y) ? x : y;
}
```

Goes beyond showing freedom from runtime errors,

Semantic Correctness

Showing that a program computes what it is supposed to:

```
/*@
  ensures ( x > y ==> \result == x );
  ensures ( x <=y ==> \result == y );
*/
int maxint (int x, int y) {
  return (x > y) ? x : y;
}
```

Goes beyond showing freedom from runtime errors, but actually the two are orthogonal (although not independent): a program can be free from runtime errors yet “incorrect”, and a program can be correct relative to freedom from runtime errors (e.g. ”the correct result is computed if there is no overflow error”).

Semantic Correctness (2)

The previous program is somewhat trivial, the code and the `ensures` clauses are very close together and the former is even simpler than the latter.

Another example . . .

Semantic Correctness (3)

```
/*@
  ensures (*ps)*a + (*pt)*b == \result;
*/
int main(int a, int b, int* ps, int* pt)
{
  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;
    }
  *ps = s; *pt = t;
  return x;
}
```

Semantic Correctness (4)

This program computes the greatest common divisor with Bézout coefficients. It is a complex program but the description of what it computes is simple. That's how it should be.

Semantic Correctness (4)

This program computes the greatest common divisor with Bézout coefficients. It is a complex program but the description of what it computes is simple. That's how it should be.
Its correctness relies on the invariant

$$s \cdot a + t \cdot b = x \quad \wedge \quad u \cdot a + v \cdot b = y$$

See TP.

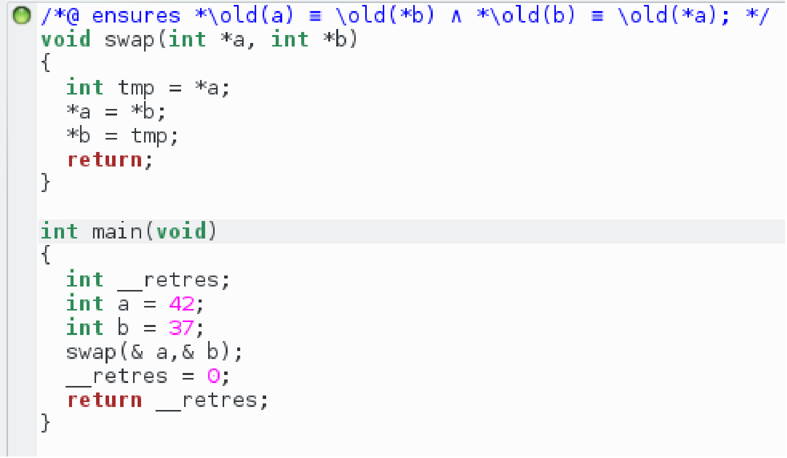
Pointers

Pointers are important in C but the main source of critical bugs.

```
/*@
ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
int tmp = *a;
*a = *b;
*b = tmp;
}

int main(){
int a = 42;
int b = 37;
swap(&a, &b);
return 0;
}
```

Works fine . . .

Analysis of swap



```
/*@ ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a); */
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}

int main(void)
{
  int __retres;
  int a = 42;
  int b = 37;
  swap(& a,& b);
  __retres = 0;
  return __retres;
}
```

Observe `\old(*a)` vs. `*\old(a)`.

Runtime Errors

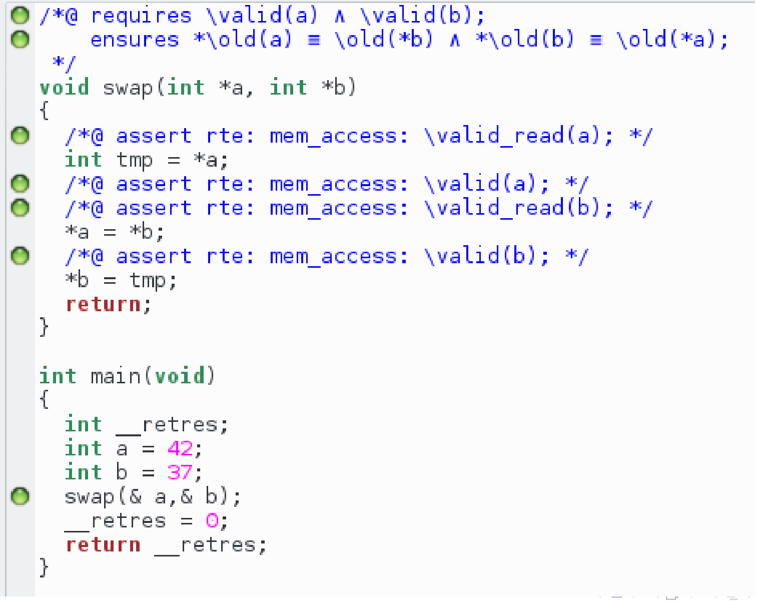
The postcondition of `swap` is proven, but if we want to prove absence of runtime errors (adding option `wp-rte`), we get lots of yellow dots.
Reason: pointers might not be valid.

Runtime Errors

The postcondition of swap is proven, but if we want to prove absence of runtime errors (adding option wp-rte), we get lots of yellow dots.
Reason: pointers might not be valid.
Solution:

```
/*@
requires \valid(a) && \valid(b);
ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b)
...
```

Analysis of swap for Runtime Errors



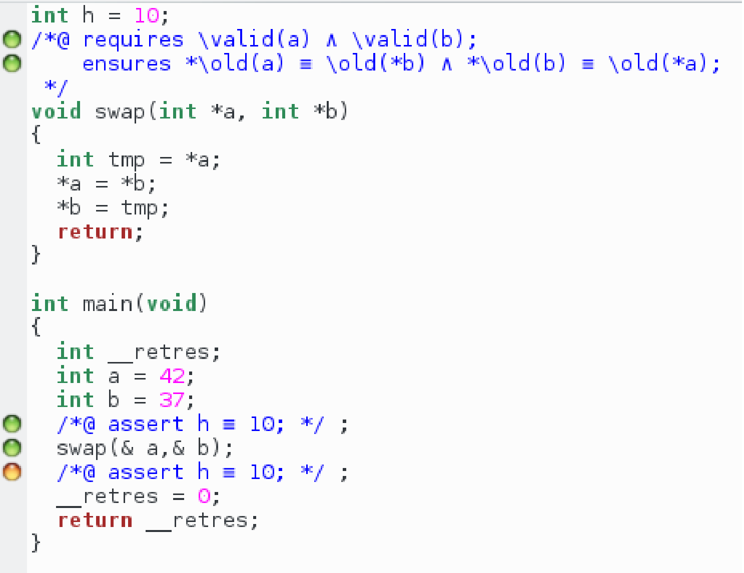
```
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a);
 */
void swap(int *a, int *b)
{
  /*@ assert rte: mem_access: \valid_read(a); */
  int tmp = *a;
  /*@ assert rte: mem_access: \valid(a); */
  /*@ assert rte: mem_access: \valid_read(b); */
  *a = *b;
  /*@ assert rte: mem_access: \valid(b); */
  *b = tmp;
  return;
}

int main(void)
{
  int __retres;
  int a = 42;
  int b = 37;
  swap(& a,& b);
  __retres = 0;
  return __retres;
}
```

Side Effects of Functions with Pointers

```
int h = 10;
/*@
requires \valid(a) && \valid(b);
ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
int tmp = *a;
*a = *b;
*b = tmp;
}

int main(){
int a = 42;
int b = 37;
//@assert h == 10;
swap(&a, &b);
//@assert h == 10;
return 0;
}
```

Assertion should be proven both times, right?

Analysis of swap with Potential Side Effects

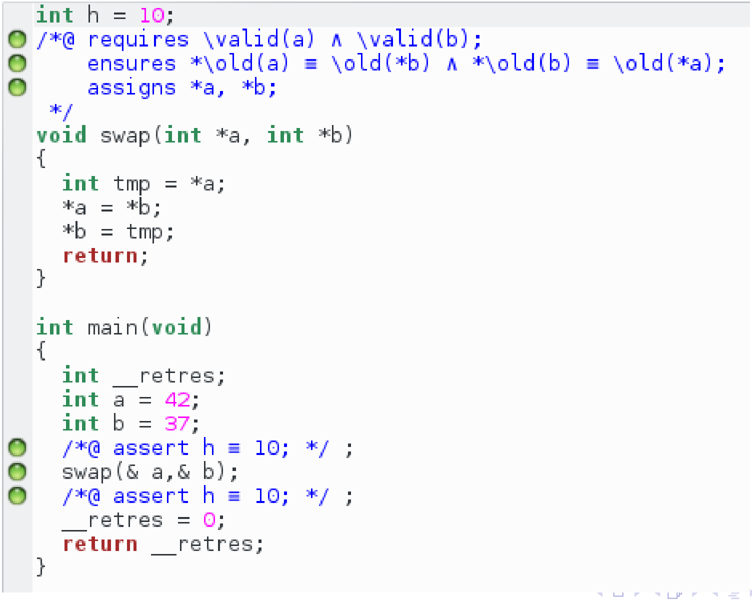


```
int h = 10;
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a);
 */
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}

int main(void)
{
  int __retres;
  int a = 42;
  int b = 37;
  /*@ assert h ≡ 10; */ ;
  swap(& a,& b);
  /*@ assert h ≡ 10; */ ;
  __retres = 0;
  return __retres;
}
```

Declaring Absence of Side Effects for swap

```
int h = 10;
/*@
requires \valid(a) && \valid(b);
assigns *a, *b;
ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
int tmp = *a;
*a = *b;
*b = tmp;
}

int main(){
int a = 42;
int b = 37;
//@assert h == 10;
swap(&a, &b);
//@assert h == 10;
return 0;
}
```

Analysis of swap: Provably no Side Effects



```
int h = 10;
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a);
    assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}

int main(void)
{
  int __retres;
  int a = 42;
  int b = 37;
  /*@ assert h ≡ 10; */ ;
  swap(& a,& b);
  /*@ assert h ≡ 10; */ ;
  __retres = 0;
  return __retres;
}
```

logic and axiomatic

To link a function definition (algorithm) to the math behind it, `logic` and `axiomatic` are useful:

```
axiomatic Euclid {
  logic integer gcd(integer a,integer b);
  axiom gcd_zero: \forall integer a; gcd(a,0) == a ;
  axiom gcd_rem:  \forall integer a,b; b!=0 ==>
                      (gcd(a,b) == gcd(b,a%b)) ;
}
```

- `axiomatic` states that the enclosed logic formulas are assumed to be always true. This allows us to inject some mathematical wisdom, but of course it must be done carefully!
- `logic` starts a definition that can only be used in specifications. Overflow is not an issue here!

See TP

Proving Incorrect Assertions

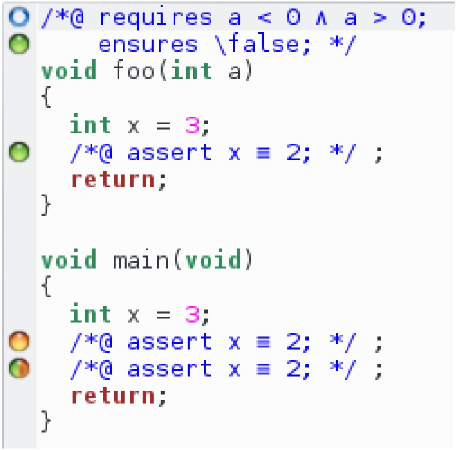
Consider:

```
/*@
requires a < 0 && a > 0;
ensures \false;
*/
void foo(int a){
  int x = 3;
  //@assert x==2;
}

void main(){
  int x = 3;
  //@assert x==2;
  //@assert x==2;
}
```

Will the postconditions and assertions be proven?

Proving Incorrect Assertions (2)

With a false precondition, anything can be proven!



```
/*@ requires a < 0 ∧ a > 0;
    ensures \false; */
void foo(int a)
{
  int x = 3;
  /*@ assert x ≡ 2; */ ;
  return;
}

void main(void)
{
  int x = 3;
  /*@ assert x ≡ 2; */ ;
  /*@ assert x ≡ 2; */ ;
  return;
}
```

Loop invariants revisited

```
int a[10];
/*@
  loop invariant 0 <= i <= 10;
  loop invariant \forall integer j; 0 <= j < i ==> a[j] == j;
  loop assigns i, a[0 .. ???];
*/
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.
- `loop assigns` is special case of invariant stating which memory locations might be modified by the loop. It is useful for the same reasons as for functions: a loop might have "side effects".
- Particularly useful for deductive verification.

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 assigned. Why true at beginning?
- Modification: Assuming that `i` and "certain" cells of the array have been modified in the previous iteration, "certain" cells will have been modified in the next iteration (which will have to be expressed relative to the new value of i).
  TP: Make this more precise!

Loop variants

- Termination can be ensured through the special clause `loop variant`.
- Requires an integer expression that strictly decreases during a loop step while remaining positive.
- For the previous example: TP exercise!

External solver

SMT solver ALT-Ergo.

SMT = SAT modulo theory.

SAT = Boolean satisfiability solving.

External solver

SMT solver ALT-Ergo.
SMT = SAT modulo theory.
SAT = Boolean satisfiability solving.
Example: $(A \wedge B) \vee (\neg A \wedge C)$ is satisfiable so SAT-solver says ok, but for $A = x > 2, B = x < 0, C = x \geq 4$, in the *arithmetic theory* $(A \wedge B) \vee (\neg A \wedge C)$ is not satisfiable, so SMT-solver will say no.

Bibliography I

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

Allan Blanchard.
Introduction to C program proof using Frama-C and its wp plugin, 2019.

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.

Bibliography II

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.

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.

Bibliography III

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.

K. Rustan M. Leino.
Efficient weakest preconditions.
Inf. Process. Lett., 93(6):281–288, 2005.