Interval Temporal Logic
A not so short introduction

Antonio Cau

Website ITL course
Slides ITL course

Part I: Introduction (2)

Overview

Features of ITL

ITL's Influence

Part II: Propositional Logic (3)

Introduction

Syntax Propositional Logic

Examples Propositional Logic

Formal semantics of Propositional Logic

Semantic Boolean Operators

Semantics of Propositional Logic

Satisfiable and valid

Summary

Exercises

Part III: Propositional ITL (4)

Introduction

Syntax Propositional ITL

Semantic Preliminaries

Semantics of Propositional ITL

Satisfiable and valid

Summary

Exercises

Part IV: First Order Logic (5)

- Introduction
- Syntax First Order Logic
- Formal semantics of First Order Logic
- Semantics of Expressions
- Semantics of Formulae
- Satisfiable and valid
- Summary
- Exercises

Part V: First Order ITL (6)

- Introduction
- Integer variable and intervals
- Examples First Order ITL
- Tempura
- Syntax First Order ITL
- Semantics of Expressions
- Semantics of Formulae
- Satisfiable and valid
- Summary
- Exercises

Part VI: Tempura (7)

- Introduction
- Justification
- Tempura Syntax
- Locations
- Expressions
- Statements
- Liquid Tank Controller
- Robot Control System
- Alternating Bit Protocol
- Summary
- Exercises

Part VII: AnaTempura (8)

- Introduction
- Runtime verification
- Application
- Runtime Verification Research
- Summary
- Exercises

Part I

Introduction

Overview

Overview

Features of ITL

ITL's Influence

Overview

This course will give a not so short introduction to Interval Temporal Logic (ITL)

ITL is a

- discrete, linear temporal logic
- for both finite and infinite intervals (sequences of states) which includes
- a basic construct for sequential composition and
- an analog of Kleene star (regular expressions)

Features of ITL

- Combines temporal logic, automata and regular expressions
- Modular reasoning about time (e.g., hardware, software, multimedia)
- Flexible notation for discrete linear order
- Analytical framework for going from infinite-time behaviour to finite-time behaviour.
- Supports sequential operators found in programs, etc.

Features of ITL (13)

- Compositionality with assumptions and commitments
- Refinement to derive concrete programs from abstract specifications.
- ITL with memory and framing can embed various kinds of imperative programs, including parallel ones.
- Executable specifications and imperative subsets.
- Temporal projection between different time granularities; enables imperative constructs for interleaved processes.

ITL's Influence (14)

- ITL helped influence Prof Mike Gordon, FRS to switch from LCF to HOL in his theorem proving tools. (E.g., see his article "From LCF to HOL: A short history").
- Hybrid systems: Duration Calculus (real and discrete time).
- ITL is used to give semantics to Verilog (EPSRC project with Oxford University).
- ITL is used in hardware/software codesign (EPSRC projects with University of Newcastle upon Tyne and Oxford University).
- ITL is used for security and trust policies (DTC-DIF projects).
- ITL used in the projects Safemos (UK) and ProCoS (EU).
- Mexitl (Kent, Winnipeg) is based on ITL, Multimedia in Executable Interval Temporal Logic.

ITL's Influence (15)

- AnaTempura runtime verification tool uses executable subset of ITL.
- Influenced Verisity Ltd.'s language temporal e (part of IEEE Standard 1647). Verisity acquired by Cadence Design Systems, Inc., a major supplier of electronic design tools and services.
- Used in the KIV theorem prover (University of Augsburg, Germany) - e.g., for Statecharts, UML and medical protocols.
- ITL with framing and temporal projection extensively studied by Duan and group (Xidian University, Xi'an, China).

(16)

Part II

Propositional Logic

Overview (17)

Introduction

Syntax Propositional Logic

Examples Propositional Logic

Formal semantics of Propositional Logic

Semantic Boolean Operators

Semantics of Propositional Logic

Satisfiable and valid

Summary

Exercises

Logic (18)

Logic is a tool for careful reasoning.

- Software and hardware development
  - Formal logic can lead to defect-free products
  - Heavily used in chip design (since Pentium disaster)
  - Used in safety-critical software design
  - Positive influence on all software development
- Foundation for programming languages (PL)
  - Together with Functions (Lambda calculus) form the semantic base for all PLs
  - Type checking and type inference - basis in logic
  - Computation model - limits of expressiveness
- Artificial intelligence, database systems
  - Grounded in formal logic

Propositional Logic (19)

- Proposition - atomic entity, true or false
  - letters $P, Q, \ldots$ denote propositional variables standing for specific propositions
    $P, Q, \ldots$ specific propositions - value: true or false
  - letters $f, f_1, f_2, \ldots$ are meta-variables denoting propositional variables or formulae
    $f, f_1, f_2, \ldots$ value: proposition, such as P, $(P \wedge Q)$
- Propositional Calculus
  - Scheme for calculating with logic formulae
- Three different propositional calculi
  - semantics-based reasoning - truth table
  - syntax-based reasoning - inference rules
  - equational reasoning - Boolean algebra

Logical Operators (20)

Logical operators provide ways to combine propositions for calculation. These are:

- $\wedge$ (And)
  - $P \wedge Q, (P \wedge Q) \wedge R, \ldots$
- $\vee$ (Or)
  - $P \vee Q, P \vee (Q \vee R), \ldots$
- $\neg$ (Not)
  - $\neg P, ((\neg P) \wedge Q), \ldots$
- $\supset$ (Implication)
  - $P \supset Q, (P \wedge Q) \supset Q, \ldots$
- $\equiv$ (Equivalence)
  - $P \equiv Q, (P \wedge Q) \equiv (Q \wedge P), \ldots$

Propositional Logic (21)

Summary of the syntax of Propositional Logic:

- Boolean values: true, false
- Boolean variables: $P, Q, \ldots$
- meta-variables: $f, f_1, f_2, \ldots$
- Boolean operators:

$\wedge$	(And),
$\vee$	(Or),
$\neg$	(Not),
$\supset$	(Implication),
$\equiv$	(Equivalence)

Propositional Logic (22)

Syntax of propositional formulae in BNF (grammar):

$$f ::= \text{true} \mid P \mid \neg f \mid f_1 \wedge f_2$$

Derived Boolean operators:

$$\begin{array}{lcl} \text{false} & \triangleq & \neg\text{true} \\ f_1 \vee f_2 & \triangleq & \neg(\neg f_1 \wedge \neg f_2) \\ f_1 \supset f_2 & \triangleq & \neg f_1 \vee f_2 \\ f_1 \equiv f_2 & \triangleq & (f_1 \supset f_2) \wedge (f_2 \supset f_1) \end{array}$$

Propositional Logic Examples (23)

Example 1

true

$\neg P$

true $\wedge P$

false $\vee \neg P$

false $\supset P$

$P \equiv (P \wedge R)$

Propositional Logic Examples (24)

Example 2

Let P denote the proposition "The moon is made of cheese"
Let Q denote the proposition "The moon is red"
The sentence "If the moon is red, it is not made of cheese" is translated as propositional formula $Q \supset (\neg P)$

Truth Table: And

(25)

P	Q	$P \wedge Q$
false	false	false
false	true	false
true	false	false
true	true	true

true iff (if and only if) **both** operands are true

Truth Table: Or

(26)

P	Q	$P \vee Q$
false	false	false
false	true	true
true	false	true
true	true	true

true iff (if and only if) **either** operands are true

Truth Table: Not

(27)

P	$\neg P$
false	true
true	false

true iff operand is false.

Truth Table: Implication

(28)

P	Q	$P \supset Q$
false	false	true
false	true	true
true	false	false
true	true	true

true iff first is true and second is true or the first is false.

Truth Table: Equivalence (29)

P	Q	$P \equiv Q$
false	false	true
false	true	false
true	false	false
true	true	true

true iff both operands have the same value.

Semantic Reasoning: Truth tables (30)

- From truth table take into account ALL its constituents. The formula $P \wedge ((\neg P) \supset Q)$ has the following constituents $\neg P$, $(\neg P) \supset Q$ as well as P and Q.

P	Q	$\neg P$	$(\neg P) \supset Q$	$P \wedge ((\neg P) \supset Q)$
false	false			
false	true			
true	false			
true	true			

- The values in the last column determine the value of the proposition:
  - if some values are true, the proposition is known to be satisfiable
  - if the values are all true, the proposition is valid
  - if all are false, the proposition is not satisfiable (contradiction).

Reasoning with Truth table (31)

- Works fine when there are 2 variables: 4 lines in the table.
- three variables: 8 lines in the table.
- 20 variables is definitely out of hand: 2^{20} lines in the table. You don't want to look at a million lines, and if you do, you will make mistakes!
- Hundreds of variables - not in a million years.

Formal Semantics (32)

Formal (denotational) semantics of Propositional Logic:

Propositions $\times$ State $\mapsto$ Bool

- Propositions: set of all possible propositions
- Bool: set of semantic Boolean values $\{tt, ff\}$
- State: a 'snapshot' of the semantic values of the propositional variables in a formula

Formal Semantics (33)

Example 3

proposition: $P \wedge (P \vee Q)$

state σ: semantic value of P is tt and of Q is ff

P	Q	$(P \vee Q)$	$P \wedge (P \vee Q)$
tt	ff	tt	tt

- State in the truth table corresponds to the first two elements in a row.
- The semantic value of the proposition w.r.t. a state is the last element in a row of the truth table.

State (34)

A state is a mapping State from the set of propositional variables Var^b to the set of Boolean values $\text{Bool} \triangleq \{\text{tt}, \text{ff}\}$.

$\text{State} : \text{Var}^b \mapsto \text{Bool}$

We will use $\sigma_0, \sigma_1, \sigma_2, \ldots$ to denote states and Σ to denote the set of all possible states.

Example 4

Let σ_0 be a state such that

$\sigma_0(P) = \text{tt}$
$\sigma_0(Q) = \text{ff}$

Semantic Boolean operators – 1 (35)

Truth tables for Semantic Boolean operators:

- not :

X	not X
tt	ff
ff	tt

Semantic Boolean operators – 2 (36)

- and :

X	Y	X and Y
tt	tt	tt
tt	ff	ff
ff	tt	ff
ff	ff	ff

- or :

X	Y	X or Y
tt	tt	tt
tt	ff	tt
ff	tt	tt
ff	ff	ff

Semantic Boolean operators – 3 (37)

- if :

X	Y	X if Y
tt	tt	tt
tt	ff	ff
ff	tt	tt
ff	ff	tt

- iff :

X	Y	X iff Y
tt	tt	tt
tt	ff	ff
ff	tt	ff
ff	ff	tt

Semantics (38)

Let $M[\![\ldots]\!](\ldots)$ be the "meaning" function from Propositions × State to $\{tt, ff\}$ and let σ_0 be a state then

$$
\begin{aligned}
M[\![true]\!](\sigma_0) &\triangleq tt \\
M[\![P]\!](\sigma_0) &\triangleq \sigma_0(P) \\
M[\![\neg f]\!](\sigma_0) &\triangleq \text{not } (M[\![f]\!](\sigma_0)) \\
M[\![f_1 \wedge f_2]\!](\sigma_0) &\triangleq (M[\![f_1]\!](\sigma_0) \text{ and } M[\![f_2]\!](\sigma_0))
\end{aligned}
$$

Semantics (39)

Example 5

Let $\sigma_0(P) = tt$ and $\sigma_0(Q) = ff$.

$$
\begin{aligned}
& M[\![P \vee Q]\!](\sigma_0) \\
= \; & M[\![\neg(\neg P \wedge \neg Q)]\!](\sigma_0) \\
= \; & \text{not } (M[\![\neg P \wedge \neg Q]\!](\sigma_0)) \\
= \; & \text{not } (M[\![\neg P]\!](\sigma_0) \text{ and } M[\![\neg Q]\!](\sigma_0)) \\
= \; & \text{not } (\text{not } (M[\![P]\!](\sigma_0)) \text{ and not } (M[\![Q]\!](\sigma_0))) \\
= \; & \text{not } (\text{not } (\sigma_0(P)) \text{ and not } (\sigma_0(Q))) \\
= \; & \text{not } (\text{not } (tt) \text{ and not } (ff)) \\
= \; & \text{not } (ff \text{ and } tt) \\
= \; & \text{not } (ff) \\
= \; & tt
\end{aligned}
$$

Satisfiable and valid (40)

- A propositional formula f is satisfiable if and only if there exists a state σ_0 such that $M[\![f]\!](\sigma_0) = tt$.
- A propositional formula f is valid if and only if for all states σ_0, $M[\![f]\!](\sigma_0) = tt$.

Example 6

- true is a valid formula.
- Proposition $P \wedge Q$ is satisfiable because $M[\![P \wedge Q]\!](\sigma_0) = tt$ where state σ_0 is such that $\sigma_0(P) = tt$ and $\sigma_0(Q) = tt$.

Summary (41)

- A propositional formula contains propositional variables and Boolean operators
- Reasoning using the truth table of a propositional formula
- The semantics (meaning) of a propositional formula is defined with respect to a state
- One can define the satisfiability and validity of a propositional formula using the semantics of a propositional formula or using the truth table

Exercises – 1 (42)

Exercise 1

Why are the following not propositional formulae? There might be more than one reason

- $P \neg Q$
- $P \wedge \supset Q$
- $P \neg \vee (Q \supset)$

Exercise 2

As seen in the course notes false, $f_1 \vee f_2$, $f_1 \supset f_2$ *and* $f_1 \equiv f_2$ *are derived propositional formulae.*

Write out $((P \vee Q) \wedge (Q \supset P))$ *using only* $\neg$ *and* $\wedge$.

Exercises – 2 (43)

Exercise 3

Give the truth table for the following propositional formulae:

- false
- $(\neg P) \vee (Q \wedge R)$
- $(\neg P) \equiv (Q \vee \neg R)$
- $(\neg P) \equiv (Q \supset R)$

Exercises – 3 (44)

Exercise 4

Let $\sigma_0(P) = tt$ *and* $\sigma_0(Q) = tt$.

Give the semantics of $P \equiv Q$, *i.e., calculate* $\mathrm{M}[\![P \equiv Q]\!](\sigma_0)$.

Exercise 5

Show that for any state σ_0 *and for propositional variables* P *and* Q *the following holds* $\mathrm{M}[\![P \vee Q]\!](\sigma_0) = (\mathrm{M}[\![P]\!](\sigma_0)$ *or* $\mathrm{M}[\![Q]\!](\sigma_0))$.

Exercises – 4

Exercise 6

Let P, Q, *and* R *be propositional variables capturing the following propositions:*

- P: *You get a first on the final exam*
- Q: *You do every exercise of the course notes*
- R: *You get a first for this module*

Write the following as formulae using P, Q, *and* R *and logical connectives.*

- *You get a first for this module, but you do not do every exercise of the course notes.*
- *To get a first for this module, it is necessary for you to get a first on the final exam.*
- *Getting a first on the final exam and doing every exercise in the course notes is sufficient for getting a first in the module.*

Exercises – 5

Exercise 7

Determine which of the following formula is satisfiable or valid. Explain why.

- false
- false $\supset$ true
- $(P \wedge Q) \equiv \neg(\neg P \vee \neg Q)$
- $(P \wedge (P \vee Q)) \equiv P$

Part III

Propositional Interval Temporal Logic

Overview

Introduction

Syntax Propositional ITL

Semantic Preliminaries

Semantics of Propositional ITL

Satisfiable and valid

Summary

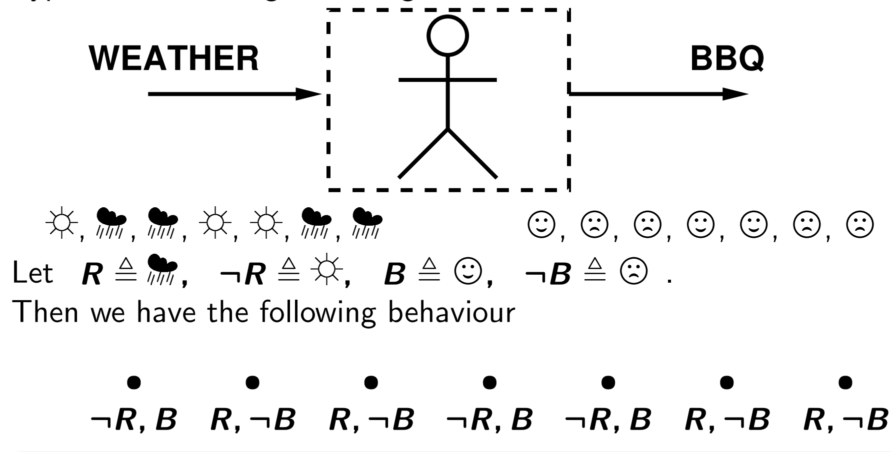
Exercises

System behaviour

We need a logic to rigorously describe the behaviour of systems.
What is a behaviour?

Example 7

Typical week in August in England:



System behaviour

- A Snapshot of system contains the variables of the system plus the values of those variables.
- A Behaviour of the system is a sequence of snapshots of the system.
- Convention: we will consider only finite sequences and only sequences with at least 1 snapshot (state).

System behaviour

How to formally describe/specify a behaviour of a system?

- Propositional logic is not adequate as it can only describe single snapshots of the system.
- One needs temporal operators to describe/specify sequences (intervals) of system snapshots
- Propositional Interval Temporal Logic can describe sequences of system snapshots using temporal operators

Skip

skip

any sequence (interval) of exactly two states (snapshots)

• •

One can combine skip with a Boolean operator for example:

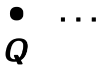
- $Q \land$ skip: any sequence (interval) of exactly two states such that in the first state Q holds

• •
Q

Propositional Formula (53)

What if you only got Q as formula?

- Q: any interval of states such that in the first state Q holds. The length of the interval is unconstrained.



What if the formula is just true?

- true: any interval of states of any length (finite and contains at least 1 state). The length of the interval is unconstrained and the states within the interval are also unconstrained.

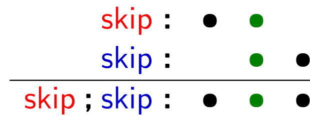
Chop (54)

; (chop)

concatenate (fuse) two intervals together such that the last state of the first interval is the same as first state of the second interval.

Example 8

skip ; skip



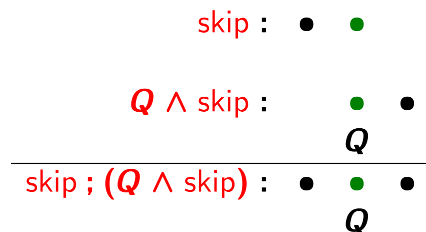
Next (55)

$\bigcirc$ (Next)

f holds from the next state onwards, $\bigcirc f \triangleq \text{skip} ; f$.

Example 9

$\bigcirc(Q \wedge \text{skip})$ using the definition: $\text{skip} ; (Q \wedge \text{skip})$



More and Empty (56)

more

interval with at least two states, $\text{more} \triangleq \bigcirc\text{true}$.

- more : • • ...

empty

interval with only one state, $\text{empty} \triangleq \neg\text{more}$.

- empty : •

Weak Next

ⓦ (Weak Next)

either an interval of only one state or f holds from the next state onwards, $ⓦ f \triangleq \text{empty} \vee \bigcirc f$.

Example 10

- $ⓦ(Q \wedge \text{skip})$ is equal to, using the definition, $\text{empty} \vee \bigcirc(Q \wedge \text{skip})$ and thus to

empty : •

$\bigcirc(Q \wedge \text{skip})$: • • •

Q

Chopstar – 1

$*$ (chopstar)

Finite number of fusion of intervals.

Example 11

- $(Q \wedge \text{skip})^*$:

  Fuse a finite number of times an interval satisfying $(Q \wedge \text{skip})$.
- Interval satisfying $Q \wedge \text{skip}$:

$Q \wedge \text{skip}$: • •

Q

Chopstar – 2

Example 11

(continued)

- 0 times fusion:

$Q \wedge \text{skip}$: • •

Q

$(Q \wedge \text{skip})^0$: •

Interval contains only 1 unconstrained state.

Chopstar – 3

Example 11

(continued)

- 1 time fusion:

$Q \wedge \text{skip}$: • •

Q

$(Q \wedge \text{skip})^1$: • •

Q

Chopstar – 4 (61)

Example 11

(continued)

- 2 times fusion:

$$
\begin{array}{rcccc}
Q \wedge \text{skip}: & \bullet & \bullet & \\
 & Q & & \\
Q \wedge \text{skip}: & & \bullet & \bullet \\
 & & Q & \\
\hline
(Q \wedge \text{skip})^2: & \bullet & \bullet & \bullet \\
 & Q & Q &
\end{array}
$$

Chopstar – 5 (62)

Example 11

(continued)

- n times fusion:

$$
\begin{array}{rcccccc}
(Q \wedge \text{skip})^n: & \bullet & \bullet & \ldots & \bullet & \bullet \\
 & Q & Q & \ldots & Q &
\end{array}
$$

- $(Q \wedge \text{skip})^*$: $\bigvee_{n=0}(Q \wedge \text{skip})^n$

Sometimes – 1 (63)

$\Diamond$ (sometimes)

there exists a suffix interval for which f holds, $\Diamond f \triangleq \text{true} \,;\, f$

Example 12

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ be an interval with 3 states.
- The suffix intervals of σ are:

$$
\begin{array}{r}
\sigma_2 \\
\sigma_1\sigma_2 \\
\sigma_0\sigma_1\sigma_2
\end{array}
$$

Sometimes – 2 (64)

Example 12

(continued)

- $\Diamond(Q)$: there exists a suffix interval for which Q holds

  We already know: Q holds for an interval iff Q is true in the first state of that interval.

  So $\Diamond(Q)$ means there exists a suffix interval σ' such that Q is true in the first state of σ'.

Sometimes – 3

Example 12

(continued)

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ then σ satisfies $\Diamond(Q)$ iff there exists a suffix interval σ' (of σ) such that Q is true in the first state of σ'.

$\sigma' =$			σ_2 • $\neg Q$	no
$\sigma' =$		σ_1 • Q	σ_2 •	yes
$\sigma' =$	σ_0 • $\neg Q$	σ_1 •	σ_2 •	no
$\sigma =$	σ_0 • $\neg Q$	σ_1 • Q	σ_2 • $\neg Q$	

Sometimes – 4

Example 12

(continued)

Are there any other intervals $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ that satisfy $\Diamond(Q)$?

Always – 1

$\Box$ (always)

for each suffix interval f holds, $\Box f \triangleq \neg\Diamond\neg f$

Example 13

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2\sigma_3$ be an interval with 4 states.
- The suffix intervals of σ are:

$$\begin{array}{r} \sigma_3 \\ \sigma_2\sigma_3 \\ \sigma_1\sigma_2\sigma_3 \\ \sigma_0\sigma_1\sigma_2\sigma_3 \end{array}$$

Always – 2

Example 13

(continued)

- $\Box(Q)$: for each suffix interval Q holds

  We already know: Q holds for an interval iff Q is true in the first state of that interval.

  So $\Box(Q)$ means for each suffix interval σ', Q is true in the first state of σ'.

Always – 3 (69)

Example 13

(continued)

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2\sigma_3$ then σ satisfies $\Box(Q)$ iff for each suffix interval σ' (of σ), Q is true in the first state of σ'.

$\sigma' =$				σ_3 • Q	yes
$\sigma' =$			σ_2 • Q	σ_3 •	yes
$\sigma' =$		σ_1 • Q	σ_2 •	σ_3 •	yes
$\sigma' =$	σ_0 • Q	σ_1 •	σ_2 •	σ_3 •	yes
$\sigma =$	σ_0 • Q	σ_1 • Q	σ_2 • Q	σ_3 • Q	

Diamond-i – 1 (70)

◇ᵢ (diamond-i)

there exists a prefix interval for which f holds, $◇_i f \triangleq f\ ;\ \text{true}$

Example 14

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ be an interval with 3 states.
- The prefix intervals of σ are:

σ_0
$\sigma_0\sigma_1$
$\sigma_0\sigma_1\sigma_2$

Diamond-i – 2 (71)

Example 14

(continued)

- $◇_i(\text{skip})$: there exists a prefix interval for which skip holds

  We already know: skip holds for an interval iff the interval has two states.

  So $◇_i(\text{skip})$ means there exists a prefix interval σ' with exactly two states.

Diamond-i – 3 (72)

Example 14

(continued)

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ then σ satisfies $◇_i(\text{skip})$ iff there exists a prefix interval σ' (of σ) with exactly two states.

$\sigma' =$	σ_0 •			no
$\sigma' =$	σ_0 •	σ_1 •		yes
$\sigma' =$	σ_0 •	σ_1 •	σ_2 •	no
$\sigma =$	σ_0 •	σ_1 •	σ_2 •	

Box-i – 1 (73)

□ⁱ (box-i)

for each prefix interval f holds, $\boxdot f \triangleq \neg \diamond \neg f$

Example 15

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2\sigma_3$ be an interval with 4 states.
- The prefix intervals of σ are:

σ_0
$\sigma_0\sigma_1$
$\sigma_0\sigma_1\sigma_2$
$\sigma_0\sigma_1\sigma_2\sigma_3$

Box-i – 2 (74)

Example 15

(continued)

- $\boxdot(Q)$: for each prefix interval Q holds

  We already know: Q holds for an interval iff Q is true in the first state of that interval.

  So $\boxdot(Q)$ means for each prefix interval σ', Q is true in the first state of σ'.

Box-i – 3 (75)

Example 15

(continued)

- Let $\sigma' \triangleq \sigma_0\sigma_1\sigma_2\sigma_3$ then σ satisfies $\boxdot(Q)$ iff for each prefix interval σ' (of σ), Q is true in the first state of σ'.

$\sigma' =$	σ_0 (•, Q)	yes
$\sigma' =$	σ_0 (•, Q) σ_1 (•)	yes
$\sigma' =$	σ_0 (•, Q) σ_1 (•) σ_2 (•)	yes
$\sigma' =$	σ_0 (•, Q) σ_1 (•) σ_2 (•) σ_3 (•)	yes
$\sigma =$	σ_0 (•, Q) σ_1 (•) σ_2 (•) σ_3 (•)	

Diamond-a – 1 (76)

◇ᵃ (diamond-a)

there exists a sub interval for which f holds, $\diamond f \triangleq true ; f ; true$

Example 16

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ be an interval with 3 states.
- The sub intervals of σ are:

σ_0
σ_1
σ_2
$\sigma_0\ \sigma_1$
$\sigma_1\ \sigma_2$
$\sigma_0\ \sigma_1\ \sigma_2$

Diamond-a – 2 (77)

Example 16

(continued)

- $\diamond_a$(empty): there exists a sub interval for which empty holds

  We already know: empty holds for an interval iff the interval has one state.

  So $\diamond_a$(empty) means there exists a sub interval σ' with exactly one state.

Diamond-a – 3 (78)

Example 16

(continued)

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ then σ satisfies $\diamond_a$(empty) iff there exists a sub interval σ' (of σ) with exactly one state.

$\sigma' =$	σ_0			yes
$\sigma' =$		σ_1		yes
$\sigma' =$			σ_2	yes
$\sigma' =$	σ_0	σ_1		no
$\sigma' =$		σ_1	σ_2	no
$\sigma' =$	σ_0	σ_1	σ_2	no
$\sigma =$	σ_0	σ_1	σ_2	

Box-a – 1 (79)

$\square_a$ (box-a)

for each sub interval f holds, $\square_a f \triangleq \neg\diamond_a\neg f$

Example 17

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ be an interval with 3 states.
- The sub intervals of σ are:

σ_0		
	σ_1	
		σ_2
σ_0	σ_1	
	σ_1	σ_2
σ_0	σ_1	σ_2

Box-a – 2 (80)

Example 17

(continued)

- $\square_a(Q)$: for each sub interval Q holds

  We already know: Q holds for an interval iff Q is true in the first state of that interval.

  So $\square_a(Q)$ means for each sub interval σ', Q is true in the first state of σ'.

- Let $\sigma \triangleq \sigma_0\sigma_1\sigma_2$ then σ satisfies $\square_a(Q)$ iff for each sub interval σ' (of σ), Q is true in the first state of σ'.

Box-a – 3 (81)

Example 17

(continued)

$\sigma' =$	σ_0 (Q)			yes
$\sigma' =$		σ_1 (Q)		yes
$\sigma' =$			σ_2 (Q)	yes
$\sigma' =$	σ_0 (Q)	σ_1		yes
$\sigma' =$		σ_1 (Q)	σ_2	yes
$\sigma' =$	σ_0 (Q)	σ_1	σ_2	yes
$\sigma =$	σ_0 (Q)	σ_1 (Q)	σ_2 (Q)	

Programming constructs (82)

skip^n, $n \geq 0$

Length of an interval, $\text{skip}^0 \triangleq \text{empty}$
$\text{skip}^n \triangleq \text{skip} ; \text{skip}^{n-1}, (n > 0)$

Example 18

- skip^3

  skip^0 : •
  skip^1 : • •
  skip^2 : • • •
  skip^3 : • • • •

Programming constructs (83)

if f_0 then f_1 else f_2

The traditional binary choice,
if f_0 then f_1 else $f_2 \triangleq (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$.

Example 19

- if A then $(\neg B \wedge \text{skip})$ else $(B \wedge \text{skip})$

  $A \wedge \neg B \wedge \text{skip}$: • ($A, \neg B$) •

  $\neg A \wedge B \wedge \text{skip}$: • ($\neg A, B$) •

Programming constructs (84)

if f_0 then f_1

The traditional if then construct,
if f_0 then $f_1 \triangleq$ if f_0 then f_1 else true.

Example 20

- if A then $(\neg B \wedge \text{skip})$

  $A \wedge \neg B \wedge \text{skip}$: • ($A, \neg B$) •

  $\neg A \wedge \text{true}$: • ($\neg A$) . . .

Programming constructs (85)

fin f

f holds in the final state, fin $f \triangleq \Box(\text{empty} \supset f)$.

Example 21

- skip^2 $\wedge$ fin Q

skip^2 : • • •
skip^2 $\wedge$ fin Q : • • •
Q (third state)

Note: Q is allowed to hold in the first and second state.

Programming constructs (86)

halt f

Halt when f holds, halt $f \triangleq \Box(\text{empty} \equiv f)$.

Example 22

- skip^2 $\wedge$ halt Q

skip^2 : • • •
skip^2 $\wedge$ halt Q : • • •
$\neg Q$ $\neg Q$ Q

Note: Q is not allowed to hold in the first and second state.

Programming constructs (87)

keep f

For all sub intervals of two states f holds, keep $f \triangleq \boxdot(\text{skip} \supset f)$.

Example 23

- skip^2 $\wedge$ keep Q

skip^2 : • • •
skip^2 $\wedge$ keep Q : • • •
Q Q (first and second states)

Programming constructs (88)

while f_0 do f_1

Traditional while loop, while f_0 do $f_1 \triangleq (f_0 \wedge f_1)^* \wedge$ fin $\neg f_0$.

Example 24

- skip^2 $\wedge$ while Q do skip

skip^2 : • • •
skip^2 $\wedge$ while Q do skip : • • •
Q Q $\neg Q$

Programming constructs (89)

repeat f_0 until f_1

Traditional repeat loop, repeat f_0 until $f_1 \triangleq f_0$; (while $\neg f_1$ do f_0).

Example 25

- skip^2 $\wedge$ repeat skip until Q

skip^2 :	•	•	•
skip^2 $\wedge$ repeat skip until Q :	•	•	•
		$\neg Q$	Q

Propositional ITL (90)

Summary of the syntax of Propositional ITL:

- Boolean values: true, false
- Boolean state variables: $P, Q, \ldots$
- Boolean operators: $\wedge, \vee, \neg, \supset, \equiv$
- Temporal operators:

skip	(skip),
;	(chop),
$\bigcirc$	(next),
$\Box$	(always),
$\Diamond$	(sometimes),
*	(chopstar),
...	

Propositional ITL (91)

Syntax of Propositional ITL in BNF:

$$f ::= \text{true} \mid Q \mid \neg f \mid f_1 \wedge f_2 \mid \text{skip} \mid f_1 ; f_2 \mid f^*$$

Derived ITL operators:

$$\bigcirc f \triangleq \text{skip} ; f$$
$$\Diamond f \triangleq \text{true} ; f$$
$$\Box f \triangleq \neg(\Diamond \neg f)$$

...

Interval and Length (92)

An interval σ is a finite sequence of states

$$\sigma : \sigma_0\sigma_1\sigma_2 \ldots$$

Let Σ^+ denote the set of all finite intervals with at least 1 state.

The length of an interval σ is denoted by $|\sigma|$ and is the number of states minus 1.

Example 26

$\sigma = \sigma_0$	$\lvert\sigma\rvert = 0$
$\sigma = \sigma_0\sigma_1$	$\lvert\sigma\rvert = 1$
$\sigma = \sigma_0\sigma_1 \ldots \sigma_n$	$\lvert\sigma\rvert = n$

Prefix, Suffix and Sub Interval (93)

Let $\sigma = \sigma_0\sigma_1\sigma_2\ldots\sigma_{|\sigma|}$ be an interval then

- $\sigma_0\ldots\sigma_k$ (where $0 \leq k \leq |\sigma|$) denotes a prefix interval of σ
- $\sigma_k\ldots\sigma_{|\sigma|}$ (where $0 \leq k \leq |\sigma|$) denotes a suffix interval of σ
- $\sigma_k\ldots\sigma_l$ (where $0 \leq k \leq l \leq |\sigma|$) denotes a sub interval of σ

Example 27

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3$ be an interval then

$\sigma_0\sigma_1$	is a prefix interval of σ
$\sigma_1\sigma_2\sigma_3$	is a suffix interval of σ
$\sigma_1\sigma_2$	is a sub interval of σ

Semantics of PITL (94)

Let $\mathsf{M}[\![\ldots]\!](\ldots)$ be the "meaning" function from PITL $\times\, \Sigma^+$ to $\{tt, ff\}$ and let σ be an interval ($\sigma \in \Sigma^+$) then

$$
\begin{array}{lcl}
\mathsf{M}[\![\mathrm{true}]\!](\sigma) & \triangleq & tt \\
\mathsf{M}[\![Q]\!](\sigma) & \triangleq & \sigma_0(Q) \\
\mathsf{M}[\![\neg f]\!](\sigma) & \triangleq & \text{not } (\mathsf{M}[\![f]\!](\sigma)) \\
\mathsf{M}[\![f_1 \wedge f_2]\!](\sigma) & \triangleq & \mathsf{M}[\![f_1]\!](\sigma) \text{ and } \mathsf{M}[\![f_2]\!](\sigma) \\
\mathsf{M}[\![\mathrm{skip}]\!](\sigma) & \triangleq & (|\sigma| = 1)
\end{array}
$$

Semantics of PITL (95)

The semantics of 'chop' is as follows $\mathsf{M}[\![f_1 \,;\, f_2]\!](\sigma) = tt$ iff

$$(\text{exists } k, \text{ s.t. } \mathsf{M}[\![f_1]\!](\sigma_0\ldots\sigma_k) = tt \text{ and } \mathsf{M}[\![f_2]\!](\sigma_k\ldots\sigma_{|\sigma|}) = tt)$$

$$
\begin{array}{ccccc}
| \leftarrow & f_1 & \rightarrow | \leftarrow & f_2 & \rightarrow | \\
\sigma_0 & & \sigma_k & & \sigma_{|\sigma|} \\
\bullet & \ldots & \bullet & \ldots & \bullet
\end{array}
$$

Interval σ is a fusion of two intervals $\sigma_0\ldots\sigma_k$ (satisfies f_1) and $\sigma_k\ldots\sigma_{|\sigma|}$ (satisfies f_2). State σ_k is shared by both.

Semantics of PITL – 1 (96)

Example 28

Let σ be an interval $\sigma_0\sigma_1\sigma_2\sigma_3$ (4 states).

We evaluate

$\mathsf{M}[\![\mathrm{skip} \,;\, (Q \wedge \mathrm{skip}^2)]\!](\sigma)$

$$
\begin{array}{l}
(\text{exists } k, \text{ s.t. } \mathsf{M}[\![\mathrm{skip}]\!](\sigma_0\ldots\sigma_k) = tt \text{ and} \\
\qquad\qquad \mathsf{M}[\![Q \wedge \mathrm{skip}^2]\!](\sigma_k\ldots\sigma_3) = tt)
\end{array}
$$

As $\mathsf{M}[\![\mathrm{skip}]\!](\sigma_0\ldots\sigma_k) = tt$ means that $|\sigma_0\ldots\sigma_k| = 1$.

We know that an interval of length 1 has two states, i.e., k can only be 1.

Semantics of PITL – 2 (97)

Example 28

(continued)

We now evaluate $M[\![Q \wedge \text{skip}^2]\!](\sigma_1 \ldots \sigma_3)$

$(M[\![Q \wedge \text{skip}^2]\!](\sigma_1 \ldots \sigma_3) = tt)$ iff
$M[\![Q]\!](\sigma_1 \ldots \sigma_3) = tt$ and $M[\![\text{skip}^2]\!](\sigma_1 \ldots \sigma_3) = tt$

Evaluate $M[\![Q]\!](\sigma_1 \ldots \sigma_3)$

$M[\![Q]\!](\sigma_1 \ldots \sigma_3) = tt$ iff $\sigma_1(Q) = tt$

Note, the definition states that we evaluate Q in the first state of the interval $\sigma_1 \ldots \sigma_3$, i.e., state σ_1.

Semantics of PITL – 3 (98)

Example 28

(continued)

Finally we evaluate $M[\![\text{skip}^2]\!](\sigma_1 \ldots \sigma_3)$

$M[\![\text{skip}^2]\!](\sigma_1 \ldots \sigma_3) = tt$ iff $|\sigma_1 \ldots \sigma_3| = 2$

An interval of length 2 means $\sigma_1 \ldots \sigma_3$ has 3 states and it certainly does.

Semantics of PITL – 4 (99)

Example 28

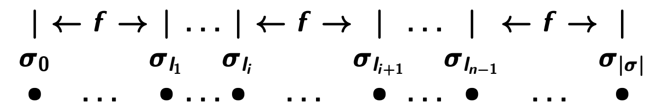
(continued)

- So $M[\![\text{skip} ; (Q \wedge \text{skip}^2)]\!](\sigma)$ evaluates to tt when we have an overall interval with 4 states (prefix interval with 2 states and a suffix interval with 3 states).
- Furthermore the value of Q in the second state (state σ_1) should be tt.

Semantics of PITL (100)

The semantics of 'chopstar' is as follows $M[\![f^*]\!](\sigma) = tt$ iff

(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
for all $0 \leq i < n, l_i \leq l_{i+1}$ and $M[\![f]\!](\sigma_{l_i} \ldots \sigma_{l_{i+1}}) = tt$)



Finite interval σ is the fusion of a finite number of finite sub-intervals each satisfying f.

Satisfiable and valid

- A propositional ITL formula f is satisfiable if and only if there exists an interval σ such that $\mathsf{M}[\![f]\!](\sigma) = \mathrm{tt}$.
- A propositional ITL formula f is valid if and only if for all intervals σ, $\mathsf{M}[\![f]\!](\sigma) = \mathrm{tt}$.

Example 29

- $(f \,;\, \mathrm{empty}) \equiv (\mathrm{empty} \,;\, f)$ is a valid formula.
- Formula $(P \wedge \mathrm{empty})\,;\,\mathrm{skip}\,;\,(P \wedge \mathrm{empty})$ is satisfiable because $\mathsf{M}[\![(P \wedge \mathrm{empty})\,;\,\mathrm{skip}\,;\,(P \wedge \mathrm{empty})]\!](\sigma) = \mathrm{tt}$ where interval σ is $\sigma_0\sigma_1$ and $\sigma_0(P) = \mathrm{tt}$ and $\sigma_1(P) = \mathrm{tt}$.

Summary

- An ITL formula contains propositional variables, Boolean operators and temporal operators.
- The semantics (meaning) of an ITL formula is defined with respect to an interval (non-empty sequence of states).
- One can define the satisfiability and validity of an ITL formula.

Exercises – 1

Exercise 8

Let $\sigma = \sigma_0\sigma_1\sigma_2\sigma_3$

- *Give all prefix intervals of* σ
- *Give all suffix intervals of* σ
- *Give all sub-intervals of* σ

Exercise 9

Give the informal semantics (picture) of following formulae:

- $\mathrm{empty} \wedge P \wedge ((P \wedge Q) \vee (\neg P \wedge R))$
- $\mathrm{empty} \wedge \neg P \wedge ((P \wedge Q) \vee (\neg P \wedge R))$
- $\mathrm{skip}^2 \wedge \boxed{a}(\mathrm{skip} \supset P)$

Exercises – 2

Exercise 10

Give the informal semantics (picture) of following formulae:

- $\mathrm{skip}^3 \wedge \mathrm{keep}\ P \wedge \mathrm{fin}\,(\neg P) \wedge \mathrm{while}\ P\ \mathrm{do}\ (Q \wedge \mathrm{skip})$
- $\mathrm{skip}^3 \wedge \Box\, P \wedge \mathrm{while}\ P\ \mathrm{do}\ (Q \wedge \mathrm{skip})$
- $\mathrm{skip}^3 \wedge \mathrm{keep}\,(\neg Q) \wedge \mathrm{fin}\,(Q) \wedge \mathrm{repeat}\,(P \wedge \mathrm{skip})\ \mathrm{until}\ Q$
- $\mathrm{skip}^3 \wedge \Box(Q) \wedge \mathrm{repeat}\,(P \wedge \mathrm{empty})\ \mathrm{until}\ Q$

Exercise 11

Give the informal semantics of the following formulae:

- $\Box(\mathrm{skip} \supset (\bigcirc P))$
- $\boxed{i}(\mathrm{skip} \supset (\bigcirc P))$
- $\boxed{a}(\mathrm{skip} \supset (\bigcirc P))$

Exercises – 3 (105)

Exercise 12

Give the formal semantics of the following formulae:

- $P \wedge Q$
- skip^2
- $\text{skip} ; P$

Exercise 13

Give the informal semantics of the following formulae:

- $\Box(\text{more} \supset (P \wedge \bigcirc Q))$
- $\boxed{\scriptstyle i}(\text{more} \supset (P \wedge \bigcirc Q))$
- $\boxed{\scriptstyle a}(\text{more} \supset (P \wedge \bigcirc Q))$

Exercises – 4 (106)

Exercise 14

Give for the following intervals their corresponding propositional ITL formulae

1. • • •
2. • • with labels P, $\neg P$
3. • • • with labels $P, \neg Q$; P, Q; $P, \neg Q$
4. • • • with labels P; P; $P, \neg Q$

Exercises – 5 (107)

Exercise 15

Give an English description of the set intervals that correspond to the following propositional formulae

- $ⓦ P \wedge \neg P \wedge \text{skip}$
- $ⓦ P \wedge \text{empty}$
- $(\Box P) ; \Diamond Q$
- $\text{more} \wedge \boxed{\scriptstyle i}\, P$
- $\text{more} \wedge \boxed{\scriptstyle a}\, P$
- $P \wedge \text{fin } P$
- $P \wedge \text{more} \wedge \text{halt } Q$
- skip^*

Exercises – 6 (108)

Exercise 16

Given following informal specification give its corresponding propositional ITL formula

All intervals of up to length 10 where in the initial state the value of variable P *is* true *and the value of variable* Q *is* false. *The value of* P *in each next state is the complemented value of* P *in the current state whereas the value of* Q *in each next state is the logical or of the value of* P *in the next state and the current value of* Q.

Exercises – 7

Exercise 17

Determine which of the following formula is satisfiable or valid. Explain why.

- $(\text{skip} ; \text{empty}) \equiv (\text{empty} ; \text{skip})$
- $\text{skip} ; \text{skip} ; \text{skip} ; (P \wedge \text{empty})$
- $((P \wedge \text{empty}) ; f) \equiv (P \wedge f)$
- $\Diamond(\text{true})$
- $\Box(\text{true})$

Part IV

First Order Logic

Overview

Introduction

Syntax First Order Logic

Formal semantics of First Order Logic

Semantics of Expressions

Semantics of Formulae

Satisfiable and valid

Summary

Exercises

Quantifiers – 1

- Quantifiers extend the expressive power of the logical notation but can be motivated as abbreviations.
- Existential quantifier. This is denoted by $\exists$ and is read as "there exists a..."

  $\exists I \in \{7, 8, 9\} \bullet \text{IsPrime}(I)$

  This quantified expression can be read as:

  there exists a value in the set $\{7, 8, 9\}$ *which satisfies the truth-valued function* IsPrime.
- The I is the bound identifier, the $I \in \{ \ldots \}$ is the constraint and the expression after the dot is the body of the quantified expression.

Quantifiers – 2 (113)

- Any free occurrences of the bound identifier within the body become bound in the quantified expression. All such occurrences refer to the bound identifier. The quantifiers thus provide a way of defining a context for free identifiers.
- The expression

  $\exists I \in \{11, 12, 13\} \bullet \text{IsOdd}(I)$

  is indeed true as it is equivalent to the proposition:

  $\text{IsOdd}(11) \vee \text{IsOdd}(12) \vee \text{IsOdd}(13)$

Quantifiers – 3 (114)

- The reason that quantifiers extend the expressive power of the logic is that the sets in the constraint of a quantified expression can be infinite. Such an expression abbreviate a disjunction which could never be completely written. For example:

  $\exists I \in \mathbb{N}_1 \bullet \text{IsPrime}(I)$

  or:

  $\exists I \in \mathbb{N}_1 \bullet \neg\text{IsPrime}(2^I - 1)$

  express facts about prime numbers.

Quantifiers – 4 (115)

- Universal Quantifiers. Denoted by ∀ and is read as "for all...".
- Just as a disjunction can be viewed as an existentially quantified expression, a conjunction such as:

  $\text{IsEven}(2) \wedge \text{IsEven}(4) \wedge \text{IsEven}(8)$

  can be written as a universally quantified expression:

  $\forall I \in \{2, 4, 8\} \bullet \text{IsEven}(I)$
- here again, universal quantification increase the expressive power of the logic when we deal with infinite sets:

  $\forall I \in \mathbb{N} \bullet \text{IsEven}(2 * I)$
- We can now formally define IsPrime as

  $$\text{IsPrime}(I) \triangleq \begin{array}{l} I \neq 1 \wedge \\ \forall D \in \mathbb{N}_1 \bullet D \text{ divides } I \supset D = 1 \vee D = I \end{array}$$

First Order Logic (116)

Summary of the syntax of Expressions:

- Integer values: $0, 1, \ldots,$
- Integer variable: $A, B, \ldots$
- meta variables: $ie, ie_0, ie_1, \ldots$
- Integer operators: $+, -, *, **, mod, div, \ldots$
- Boolean values: true, false
- Propositional variables: $P, Q, \ldots$
- meta variables: $be, be_0, be_1, \ldots$
- Boolean operators: $\wedge$, $\vee$, $\neg$, $\supset$, $\equiv$, $\ldots$.

Summary of the syntax of Formulae:

- Meta variables: $e, e_0, \ldots$ (integer or Boolean expressions)
- Meta variables: $f, f_0, \ldots$
- Boolean predicates: $e_1 = e_2, e_1 \neq e_2, e_1 < e_2, e_1 \leq e_2, e_1 > e_2, e_1 \geq e_2, \ldots$
- Boolean operators: $\wedge$, $\vee$, $\neg$, $\supset$, $\equiv$, $\exists V \bullet f$, $\forall V \bullet f$

  $\exists V \bullet f$ abbreviates $\exists V \in \text{Val} \bullet f$

First Order Logic (117)

Syntax of Integer Expressions in BNF:

$ie ::= \ z \mid A \mid ig(ie_1, \dots, ie_n)$
where z is an integer constant and ig an integer operator.

Syntax of Boolean Expressions in BNF:

$be ::= \ b \mid Q \mid bg(be_1, \dots, be_n)$
where b is a Boolean constant and bg an Boolean operator.

Syntax of First Order Formulae in BNF (grammar):

$f ::= \ \text{true} \mid h(e_1, \dots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall V \bullet f$
where h is a Boolean predicate over integer or Boolean expressions.

Derived formulae:

$$\exists V \bullet f \ \triangleq \ \neg \forall V \bullet \neg f$$

Truth Table (118)

Again one could use truth tables to define the semantics of first order formulae but ...

Example 30

Let $\forall A \in \mathbb{N} \bullet (A + 1 > A)$ be a formula

A	$A+1>A$
1	true
2	true
...	

In order to determine the truth value of $\forall A \in \mathbb{N} \bullet (A + 1 > A)$ one needs to consider all rows in above truth table.

Truth tables are not suitable to determine the semantics or reason about first order formulae.

Semantics of formula (119)

We will use a denotational semantics to define the formal semantics of a first order formula.

$\text{Formulae} \times \text{State} \ \mapsto \ \text{Bool}$

- Formulae: set of all possible first order formulae
- Bool: set of semantic Boolean values $\{\text{tt}, \text{ff}\}$
- State: a 'snapshot' of the semantic values of the propositional variables and integer variables in a formula

State (120)

A State is the union of

- an integer state State^e which is a mapping from the set of integer variables Var^e to the set of integer values Val and
- a Boolean state State^b which is a mapping from the set of propositional variable Var^b to the set of Boolean values Bool.

$\text{State} : (\text{Var}^e \ \mapsto \ \text{Val}) \cup (\text{Var}^b \ \mapsto \ \text{Bool})$

where $\text{Var}^e \cap \text{Var}^b = \emptyset$.

We will use $\sigma_0, \sigma_1, \sigma_2, \dots$ to denote states and Σ to denote the set of all possible states.

Example 31

Let P be a Boolean variable and A be an integer variable then σ_0 s.t. $\sigma_0(P) = \text{tt}$ and $\sigma_0(A) = 5$ is a state.

Semantics of expressions (121)

Let $\mathrm{E}[\![\ldots]\!](\ldots)$ be the "meaning" (semantic) function from Expressions $\times\ \boldsymbol{\Sigma}$ to Val (integer or Boolean values) and let $\boldsymbol{\sigma_0}$ be a state then

$$
\begin{array}{lcl}
\mathrm{E}[\![z]\!](\sigma_0) & = & z \\
\mathrm{E}[\![A]\!](\sigma_0) & = & \sigma_0(A) \\
\mathrm{E}[\![ig(ie_1,\ldots,ie_n)]\!](\sigma_0) & = & ig(\mathrm{E}[\![ie_1]\!](\sigma_0),\ldots,\mathrm{E}[\![ie_n]\!](\sigma_0)) \\
\mathrm{E}[\![b]\!](\sigma_0) & = & b \\
\mathrm{E}[\![Q]\!](\sigma_0) & = & \sigma_0(Q) \\
\mathrm{E}[\![bg(be_1,\ldots,be_n)]\!](\sigma_0) & = & bg(\mathrm{E}[\![be_1]\!](\sigma_0),\ldots,\mathrm{E}[\![be_n]\!](\sigma_0))
\end{array}
$$

Example 32

$$
\begin{array}{lcl}
\mathrm{E}[\![\text{Account}]\!](\sigma_0) & = & \sigma_0(\text{Account}) \\
\mathrm{E}[\![\text{In}]\!](\sigma_0) & = & \sigma_0(\text{In}) \\
\mathrm{E}[\![\text{Account} + \text{In}]\!](\sigma_0) & = & \mathrm{E}[\![\text{Account}]\!](\sigma_0) + \mathrm{E}[\![\text{In}]\!](\sigma_0) \\
 & = & \sigma_0(\text{Account}) + \sigma_0(\text{In})
\end{array}
$$

Semantics of formulae (122)

Let $\mathrm{M}[\![\ldots]\!](\ldots)$ be the "meaning" function from Formulae $\times\ \boldsymbol{\Sigma}$ to Bool (set of Boolean values, $\{\mathrm{tt},\mathrm{ff}\}$) and let $\boldsymbol{\sigma_0}$ be a state then

$$
\begin{array}{lll}
\mathrm{M}[\![\text{true}]\!](\sigma_0) & = & \mathrm{tt} \\
\mathrm{M}[\![h(e_1,\ldots,e_n)]\!](\sigma_0) = \mathrm{tt} & \text{iff} & h(\mathrm{E}[\![e_1]\!](\sigma_0),\ldots,\mathrm{E}[\![e_n]\!](\sigma_0)) \\
\mathrm{M}[\![\neg f]\!](\sigma_0) = \mathrm{tt} & \text{iff} & \text{not } (\mathrm{M}[\![f]\!](\sigma_0) = \mathrm{tt}) \\
\mathrm{M}[\![f_1 \wedge f_2]\!](\sigma_0) = \mathrm{tt} & \text{iff} & (\mathrm{M}[\![f_1]\!](\sigma_0) = \mathrm{tt}) \text{ and} \\
 & & (\mathrm{M}[\![f_2]\!](\sigma_0) = \mathrm{tt})
\end{array}
$$

Example 33

$$
\begin{array}{ll}
(\mathrm{M}[\![\neg(\text{Account} < 0)]\!](\sigma_0) = \mathrm{tt}) & \text{iff} \\
\text{not } (\mathrm{M}[\![\text{Account} < 0]\!](\sigma_0) = \mathrm{tt}) & \text{iff} \\
\text{not } (\mathrm{E}[\![\text{Account}]\!](\sigma_0) < \mathrm{E}[\![0]\!](\sigma_0)) & \text{iff} \\
\text{not } (\sigma_0(\text{Account}) < 0) &
\end{array}
$$

Example (123)

Example 34

$$
\begin{array}{ll}
(\mathrm{M}[\![\text{Account} = 50 \wedge \text{In} \geq 0]\!](\sigma_0) = \mathrm{tt}) & \text{iff} \\
(\mathrm{M}[\![\text{Account} = 50]\!](\sigma_0) = \mathrm{tt}) \text{ and } (\mathrm{M}[\![\text{In} \geq 0]\!](\sigma_0) = \mathrm{tt}) & \text{iff} \\
(\mathrm{E}[\![\text{Account}]\!](\sigma_0) = \mathrm{E}[\![50]\!](\sigma_0)) \text{ and } (\mathrm{E}[\![\text{In}]\!](\sigma_0) \geq \mathrm{E}[\![0]\!](\sigma_0)) & \text{iff} \\
\sigma_0(\text{Account}) = 50 \text{ and } \sigma_0(\text{In}) \geq 0 &
\end{array}
$$

Semantics of formulae (124)

Let $\sigma_0 \sim_v \sigma_0'$ denote that the states σ_0 and σ_0' are identical with the possible exception of the mapping for the variable v.

Example 35

Let σ_0 and σ_0' be states.

- Let $\sigma_0(\text{Cash}) = 0$ and $\sigma_0(\text{In}) = 2$
- Let $\sigma_0'(\text{Cash}) = 0$ and $\sigma_0'(\text{In}) = 3$

Then $\sigma_0 \sim_{\text{In}} \sigma_0'$.

Semantics of formulae (125)

The semantics of $\forall V \bullet f$ is defined in terms of $\sigma_0 \sim_V \sigma'_0$

$$\mathrm{M}[\![\forall V \bullet f]\!](\sigma_0) = \mathrm{tt} \quad \text{iff} \quad (\text{for all } \sigma'_0 \text{ s.t. } \sigma_0 \sim_V \sigma'_0, \mathrm{M}[\![f]\!](\sigma'_0) = \mathrm{tt})$$

Example 36

$\mathrm{M}[\![\forall \mathrm{In} \bullet \mathrm{In} \geq 0]\!](\sigma_0) = \mathrm{tt}$ iff
$(\text{for all } \sigma'_0 \text{ s.t. } \sigma_0 \sim_{\mathrm{In}} \sigma'_0, \mathrm{M}[\![\mathrm{In} \geq 0]\!](\sigma'_0) = \mathrm{tt})$ iff
$(\text{for all } \sigma'_0 \text{ s.t. } \sigma_0 \sim_{\mathrm{In}} \sigma'_0, \sigma'_0(\mathrm{In}) \geq 0)$

Satisfiable and valid (126)

- A first order formula f is satisfiable if and only if there exists a state σ_0 such that $\mathrm{M}[\![f]\!](\sigma_0) = \mathrm{tt}$.
- A first order formula f is valid if and only if for all states σ_0, $\mathrm{M}[\![f]\!](\sigma_0) = \mathrm{tt}$.

Example 37

- $0 < 1$ is a valid formula.
- Formula $A < B$ is satisfiable because $\mathrm{M}[\![A < B]\!](\sigma_0) = \mathrm{tt}$ where state σ_0 is such that $\sigma_0(A) = 0$ and $\sigma_0(B) = 1$.

Summary (127)

- A first order formula contains predicates, propositional variables, Boolean operators and quantifiers.
- The semantics (meaning) of a first order formula is defined with respect to a state.
- One can define the satisfiability and validity of a first order formula.

Exercises – 1 (128)

Exercise 18

Let $\mathrm{IsOdd}(X)$ *be the statement "X is an odd number".*

Express each of these first order formula in English.

- $\exists X \bullet \mathrm{IsOdd}(X)$
- $\forall X \bullet \mathrm{IsOdd}(X)$
- $\neg(\exists X \bullet \mathrm{IsOdd}(X))$
- $\neg(\forall X \bullet \mathrm{IsOdd}(X))$
- $\exists X \bullet \neg\mathrm{IsOdd}(X)$
- $\forall X \bullet \neg\mathrm{IsOdd}(X)$
- $\neg(\exists X \bullet \neg\mathrm{IsOdd}(X))$
- $\neg(\forall X \bullet \neg\mathrm{IsOdd}(X))$

Which of these "mean" the same thing?

Exercises – 2

Exercise 19

Give the formal semantics of the following formulae

- $4 > 3$
- $A + 5 \leq B \wedge B = D - 7$
- $P \equiv (A + 5 \leq B)$
- $Q \equiv (B = D - 7)$
- $\forall A \bullet (A + 1 > A)$
- $A = 8 \wedge \exists B \bullet (A = 2 ** B)$

Exercises – 3

Exercise 20

Let $p(X, Y)$ be the statement "X+Y=X-Y". If the domain for both variables is the set of integers, what are the truth values of the following?

- $p(1, 1)$
- $p(2, 0)$
- $\forall Y \bullet p(1, Y)$
- $\exists X \bullet p(X, 2)$
- $\exists X \bullet \exists Y \bullet p(X, Y)$
- $\forall X \bullet \exists Y \bullet p(X, Y)$
- $\exists Y \bullet \forall X \bullet p(X, Y)$
- $\forall Y \bullet \exists X \bullet p(X, Y)$
- $\forall X \bullet \forall Y \bullet p(X, Y)$

Exercises – 4

Exercise 21

Determine which of the following formula is satisfiable or valid. Explain why.

- $(A + B) = (B + A)$
- $\exists A \bullet A = 0$
- $\forall A \bullet \mathrm{IsOdd}(A)$
- $(\exists A \bullet f) \supset (\forall A \bullet f)$

Exercises – 5

Exercise 22

Let $p(X)$ be the statement "X has a pen", let $q(X)$ be the statement "X has a pencil", and let $r(X)$ be the statement "X has a piece of paper". Express each of these statements in first-order logic using these relations. Let the domain be your classmates.

- *A classmate has a pen, a pencil, and a piece of paper.*
- *All your classmates have a pen, a pencil, or a piece of paper.*
- *At least one of your classmates has a pen and a pencil, but not a piece of paper.*
- *None of your classmates has a pen, a pencil, and a piece of paper.*
- *For each of the three writing materials, there is a classmate of yours that has one.*

Exercises – 6

Exercise 23

Let $\boldsymbol{p(X)}$ be the statement "X is a duck", let $\boldsymbol{q(X)}$ be the statement "X is one of my poultry", let $\boldsymbol{r(X)}$ be the statement 'X is an officer", and $\boldsymbol{s(X)}$ be the statement "X is willing to waltz". Express each of these statements using quantifiers, logical connectives, and the relations $\boldsymbol{p(X), q(X), r(X)}$, and $\boldsymbol{s(X)}$.

- *No ducks are willing to waltz.*
- *No officers ever decline to waltz.*
- *All my poultry are ducks.*
- *My poultry are not officers.*
- *Does the fourth item follow from the first three taken together? Explain your answer.*

Exercises – 7

Exercise 24

Translate the following conversational English statements into first-order logic, using the suggested predicates, or inventing appropriately-named ones if none provided. (You may also freely use $=$ which we'll choose to always interpret as the standard equality relation.)

- *"All books rare and used". This is claimed by a local bookstore; what is the intended domain? Do you believe they mean to claim "all books rare or used"?*
- *"Everybody who knows that UFOs have kidnapped people knows that Agent Mulder has been kidnapped." (Is this true, presuming that no UFOs have actually visited Earth ...yet?)*

Exercises – 8

Exercise 25

The puzzle game of Sudoku is played on a $\mathbf{9 \times 9}$ grid, where each square holds a number between $\mathbf{1}$ and $\mathbf{9}$. The positions of the numbers must obey constraints. Each row and each column has each of the 9 numbers. Each of the 9 non-overlapping $\mathbf{3 \times 3}$ square sub-grids has each of the 9 numbers.

Throughout the game, some of the values have not been discovered, although they are determined. You start with some numbers revealed, enough to guarantee that the rest of the board is uniquely determined by the constraints.

Exercises – 9

Exercise 25

(continued)
So, our domain is $\mathbf{\{1, 2, 3, 4, 5, 6, 7, 8, 9\}}$. To model the game, we will use the following relations:

- value$\boldsymbol{(r, c, v)}$ *indicates that at row $\boldsymbol{r}$, column $\boldsymbol{c}$ is the value $\boldsymbol{v}$.*
- $\boldsymbol{v = w}$ *is the standard equality relation.*
- subgrid$\boldsymbol{(g, r, c)}$ *indicates that subgrid $\boldsymbol{g}$ includes the location at row $\boldsymbol{r}$, column $\boldsymbol{c}$.*

Express the row, column, and subgrid constraints for Sudoku as first order formulae and briefly explain them. In addition, you should include constraints on our above relations, such as that each location holds one value.

Part V

First Order Interval Temporal Logic

Overview

Introduction

Integer variable and intervals

Examples First Order ITL

Tempura

Syntax First Order ITL

Semantics of Expressions

Semantics of Formulae

Satisfiable and valid

Summary

Exercises

Introduction

Sofar we have only seen Boolean variables in Interval Temporal Logic. In order to describe the behaviour of systems with a 'programming language'-like logic like a we need to introduce

- Integer variables
- Temporal variables
- Quantification

Integer variable

A

integer variable, value of A in the current state

Example 38

$A = 0 \wedge (\text{skip} ; (A = 1 \wedge \text{empty}))$

$A = 0$:	• $A = 0$	...
skip :	•	•
$(A = 1 \wedge \text{empty})$:		• $A = 1$
$\text{skip} ; (A = 1 \wedge \text{empty})$:	•	• $A = 1$
$A = 0 \wedge (\text{skip} ; (A = 1 \wedge \text{empty}))$:	• $A = 0$	• $A = 1$

Temporal variable (141)

$\bigcirc A$

the value of A in the next state

Example 39

$A = 0 \land \text{skip} \land (\bigcirc A) = 1$

$$
\begin{array}{rll}
A = 0 : & \bullet_{A=0} & \ldots \\
\text{skip} \land (\bigcirc A) = 1 : & \bullet & \bullet_{A=1} \\
\hline
A = 0 \land \text{skip} \land (\bigcirc A) = 1 : & \bullet_{A=0} & \bullet_{A=1}
\end{array}
$$

$A := e$

unit assignment, $A := e \triangleq (\bigcirc A) = e$

Temporal variable (142)

fin A

the value of A in the last state

Example 40

$A = 0 \land \text{skip}^2 \land (\text{fin } A) = 1$

$$
\begin{array}{rlll}
A = 0 : & \bullet_{A=0} & \ldots & \\
\text{skip}^2 \land (\text{fin } A) = 1 : & \bullet & \bullet & \bullet_{A=1} \\
\hline
A = 0 \land \text{skip}^2 \land (\text{fin } A) = 1 : & \bullet_{A=0} & \bullet & \bullet_{A=1}
\end{array}
$$

$A \leftarrow e$

temporal assignment, $A \leftarrow e \triangleq (\text{fin } A) = e$

Derived Operator (143)

A gets e

A continually gets e, $A \text{ gets } e \triangleq \boxdot(\text{skip} \supset A \leftarrow e)$

Example 41

$$
A = 0 \land \text{skip}^2 \land A \text{ gets } A + 1 : \quad \bullet_{A=0} \quad \bullet_{A=1} \quad \bullet_{A=2}
$$

stable A

A remains stable, $\text{stable } A \triangleq A \text{ gets } A$

Quantification (144)

$\exists X \bullet f$

X acts as local variable in f

Example 42

$$
\begin{array}{rllll}
& A = 0 \land \text{skip}^2 \land A \text{ gets } A + 1 : & \bullet_{A=0} & \bullet_{A=1} & \bullet_{A=2} \\
\exists A \bullet & A = 0 \land \text{skip}^2 \land & & & \\
& A \text{ gets } A + 1 \land B = \text{fin}(A) : & \bullet_{B=2} & \bullet & \bullet
\end{array}
$$

len(n)

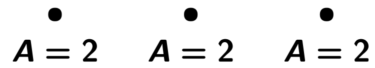
length of an interval,

$\text{len}(n) \triangleq \exists I \bullet (I = 0) \land (I \text{ gets } I + 1) \land (I \leftarrow n)$

Examples First Order ITL

From pictorial semantics to first order ITL

Example 43



1. $\mathrm{len}(2) \wedge \Box\, A = 2$
2. $\mathrm{len}(2) \wedge A = 2 \wedge (\bigcirc A) = 2 \wedge \bigcirc((\bigcirc A) = 2)$

What is the English description of the interval?

Examples First Order ITL

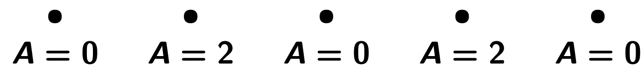
Example 44

1. $A = 0 \wedge \mathrm{while}\ A \neq 6\ \mathrm{do}\ (\mathrm{skip} \wedge A := A + 2)$
2. $\mathrm{len}(3) \wedge A = 0 \wedge A\ \mathrm{gets}\ A + 2$
3. $A = 0 \wedge A\ \mathrm{gets}\ A + 2 \wedge \mathrm{halt}\ (A = 6)$

What is the English description of the interval?

Examples First Order ITL

Example 45

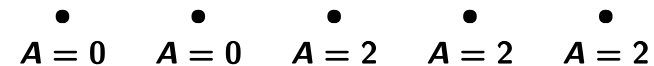


1. $\mathrm{len}(4) \wedge A = 0 \wedge A\ \mathrm{gets}\ 2 - A$
2. $((A = 0 \wedge A := 2 \wedge \mathrm{skip})\ ;\ \mathrm{skip};$
   $(A = 0 \wedge A := 2 \wedge \mathrm{skip})\ ;\ \mathrm{skip})$
   $\wedge\ \mathrm{fin}\ (A = 0)$

What is the English description of the interval?

Examples First Order ITL

Example 46



1. $(\mathrm{skip} \wedge \Box\, A = 0)\ ;\ \mathrm{skip}\ ;\ (\mathrm{len}(2) \wedge \Box\, A = 2)$

What is the English description of the interval?

Examples First Order ITL (149)

From first order ITL to pictorial semantics

Example 47

len(3) $\wedge$ $A = 1$ $\wedge$ A gets $2 + A$

$A = 1$ $A = 3$ $A = 5$ $A = 7$

What is the English description of the interval?

Examples First Order ITL (150)

Example 48

$(A = 0 \wedge A := A + 4 \wedge \text{skip})\ ;\ (\text{skip} \wedge A := 0)$

$A = 0$ $A = 4$ $A = 0$

What is the English description of the interval?

Examples First Order ITL (151)

Example 49

halt $(A = 0) \wedge A = 8 \wedge (\text{skip} \wedge A := A - 2)^*$

$A = 8$ $A = 6$ $A = 4$ $A = 2$ $A = 0$

What is the English description of the interval?

Tempura – 1 (152)

- Would it not be nice to have a tool that would generate the interval corresponding to a given ITL formula?
- There exists such a tool: Tempura

```
define test() = {
  exists A: { halt(A=0) and A=8 and chopstar(skip and A:=A-2) and
              always output(A)
  }
}.
```

Output:

```
State   0: A=8
State   1: A=6
State   2: A=4
State   3: A=2
State   4: A=0
```

Tempura – 2 (153)

A more complicated example:

```
/* run */ define miracle() =
 exists A,I : {
  list (A, 12) and stable (struct (A)) and
  forall i < |A| : { A[i] = Random mod 20 } and
  I = 0 and always (output A) and
  chopstar {
   forall i < (|A| - 1) : {
    if I = i mod 2 then {
     if A[i] > A[i + 1]
     then (A[i] := A[i + 1] and A[i + 1] := A[i])
     else (A[i] := A[i] and A[i + 1] := A[i + 1])
    }
   } and
```

Tempura – 3 (154)

```
       skip and I:=(I+1) mod 2 and
       if |A| mod 2 = 1 then {
        if I = 0
        then A[|A|-1] := A[|A|-1]
        else A[0] := A[0]
       } else {
        if I = 1 then {
         if A[0] > A[|A|-1]
         then (A[0] := A[|A|-1] and A[|A|-1] := A[0])
         else (A[0] := A[0] and A[|A|-1] := A[|A|-1])
        }
       }
      } /* chopstar */
      and halt( forall i<(|A|-1) : A[i] <= A[i + 1])
     }.
```

Tempura – 4 (155)

Output:

```
State   0: A=[10,15,3,4,10,18,16,4,4,13,11,5]
State   1: A=[10,15,3,4,10,18,4,16,4,13,5,11]
State   2: A=[10,3,15,4,10,4,18,4,16,5,13,11]
State   3: A=[3,10,4,15,4,10,4,18,5,16,11,13]
State   4: A=[3,4,10,4,15,4,10,5,18,11,16,13]
State   5: A=[3,4,4,10,4,15,5,10,11,18,13,16]
State   6: A=[3,4,4,4,10,5,15,10,11,13,18,16]
State   7: A=[3,4,4,4,5,10,10,15,11,13,16,18]
State   8: A=[3,4,4,4,5,10,10,11,15,13,16,18]
State   9: A=[3,4,4,4,5,10,10,11,13,15,16,18]
```

So why is this "program" called miracle?

First Order ITL (156)

Summary of the syntax of Expressions:

- Integer values: $\mathbf{0}, \mathbf{1}, \ldots,$
- State integer variable: $\boldsymbol{A}, \boldsymbol{B}, \ldots$
- Integer operators: $+, -, *, **, mod, div, \ldots$
- Temporal integer variable: $\bigcirc\boldsymbol{A}, \bigcirc\boldsymbol{B}, \ldots,$ fin $\boldsymbol{A}$, fin $\boldsymbol{B}, \ldots$
- Boolean values: true, false
- Boolean state variables: $\boldsymbol{P}, \boldsymbol{Q}, \ldots$
- Temporal Boolean variable: $\bigcirc\boldsymbol{P}, \bigcirc\boldsymbol{Q}, \ldots,$ fin $\boldsymbol{P}$, fin $\boldsymbol{Q}, \ldots$

Summary of the syntax of Formulae:

- Boolean predicates: $e_1 = e_2, e_1 \neq e_2, e_1 < e_2, e_1 \leq e_2,$ $e_1 > e_2, e_1 \geq e_2, \ldots$
- Boolean operators: $\wedge, \vee, \neg, \supset, \equiv, \forall V \bullet f$
- Temporal operators: skip, ;, *, $\bigcirc$, $\Box$, $\Diamond$, ...

First Order ITL (157)

Syntax of Integer Expressions in BNF:

$$ie ::= \; z \mid A \mid ig(ie_1, \ldots, ie_n) \mid \bigcirc A \mid \text{fin } A$$

where z is an integer constant and ig an integer operator.

Syntax of Boolean Expressions in BNF:

$$be ::= \; b \mid Q \mid bg(be_1, \ldots, be_n) \mid \bigcirc Q \mid \text{fin } Q$$

where b is a Boolean constant and bg a Boolean operator.

Syntax of First Order Formulae in BNF:

$$f ::= \; \text{true} \mid h(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall V \bullet f \mid \text{skip} \mid f_1 \,;\, f_2 \mid f^*$$

where h is a Boolean predicate over integer or Boolean expressions.

First Order ITL (158)

Derived formulae:

$$
\begin{array}{lcl}
\text{false} & \triangleq & \neg\text{true} \\
f_1 \vee f_2 & \triangleq & \neg(\neg f_1 \wedge \neg f_2) \\
f_1 \supset f_2 & \triangleq & \neg f_1 \vee f_2 \\
f_1 \equiv f_2 & \triangleq & (f_1 \supset f_2) \wedge (f_2 \supset f_1) \\
\exists V \bullet f & \triangleq & \neg \forall V \bullet \neg f \\
\bigcirc f & \triangleq & \text{skip} \,;\, f \\
\Diamond f & \triangleq & \text{true} \,;\, f \\
\Box f & \triangleq & \neg(\Diamond \neg f) \\
\ldots & &
\end{array}
$$

Semantics of expressions (159)

Let $\mathsf{E}[\![\ldots]\!](\ldots)$ be the "meaning" (semantic) function from Expressions $\times\, \Sigma^+$ to Val and let $\sigma = \sigma_0\sigma_1\ldots$ be an interval then

$$
\begin{array}{lcl}
\mathsf{E}[\![z]\!](\sigma) & = & z \\
\mathsf{E}[\![A]\!](\sigma) & = & \sigma_0(A) \\
\mathsf{E}[\![ig(ie_1, \ldots, ie_n)]\!](\sigma) & = & g(\mathsf{E}[\![ie_1]\!](\sigma), \ldots, \mathsf{E}[\![ie_n]\!](\sigma)) \\
\mathsf{E}[\![\bigcirc A]\!](\sigma) & = & \begin{cases} \sigma_1(A) & \text{if } |\sigma| > 0 \\ \text{choose-any-from}(\mathbb{Z}) & \text{otherwise} \end{cases} \\
\mathsf{E}[\![\text{fin } A]\!](\sigma) & = & \sigma_{|\sigma|}(A) \\
\mathsf{E}[\![b]\!](\sigma) & = & b \\
\mathsf{E}[\![Q]\!](\sigma) & = & \sigma_0(Q) \\
\mathsf{E}[\![bg(be_1, \ldots, be_n)]\!](\sigma) & = & bg(\mathsf{E}[\![be_1]\!](\sigma), \ldots, \mathsf{E}[\![be_n]\!](\sigma)) \\
\mathsf{E}[\![\bigcirc Q]\!](\sigma) & = & \begin{cases} \sigma_1(Q) & \text{if } |\sigma| > 0 \\ \text{choose-any-from}(\text{Bool}) & \text{otherwise} \end{cases} \\
\mathsf{E}[\![\text{fin } Q]\!](\sigma) & = & \sigma_{|\sigma|}(Q)
\end{array}
$$

Example of semantics (160)

Example 50

$\mathsf{E}[\![\text{Account}]\!](\sigma) = \sigma_0(\text{Account})$

$\mathsf{E}[\![\text{Account} - \text{Out}]\!](\sigma) =$
$\mathsf{E}[\![\text{Account}]\!](\sigma) - \mathsf{E}[\![\text{Out}]\!](\sigma) =$
$\sigma_0(\text{Account}) - \sigma_0(\text{Out})$

Semantics of formulae (161)

Let $\mathrm{M}[\![\ldots]\!](\ldots)$ be the "meaning" function from Formulae $\times\ \boldsymbol{\Sigma}^+$ to Bool (set of Boolean values, $\{\mathrm{tt}, \mathrm{ff}\}$) and let $\boldsymbol{\sigma} = \boldsymbol{\sigma}_0\boldsymbol{\sigma}_1\ldots$ be an interval then

$$
\begin{array}{lll}
\mathrm{M}[\![\mathrm{true}]\!](\boldsymbol{\sigma}) & = & \mathrm{tt} \\
\mathrm{M}[\![h(e_1,\ldots,e_n)]\!](\boldsymbol{\sigma}) = \mathrm{tt} & \mathrm{iff} & h(\mathrm{E}[\![e_1]\!](\boldsymbol{\sigma}),\ldots,\mathrm{E}[\![e_n]\!](\boldsymbol{\sigma})) \\
\mathrm{M}[\![\neg f]\!](\boldsymbol{\sigma}) = \mathrm{tt} & \mathrm{iff} & \mathrm{not}\ (\mathrm{M}[\![f]\!](\boldsymbol{\sigma}) = \mathrm{tt}) \\
\mathrm{M}[\![f_1 \wedge f_2]\!](\boldsymbol{\sigma}) = \mathrm{tt} & \mathrm{iff} & (\mathrm{M}[\![f_1]\!](\boldsymbol{\sigma}) = \mathrm{tt})\ \mathrm{and} \\
 & & (\mathrm{M}[\![f_2]\!](\boldsymbol{\sigma}) = \mathrm{tt}) \\
\mathrm{M}[\![\mathrm{skip}]\!](\boldsymbol{\sigma}) = \mathrm{tt} & \mathrm{iff} & |\boldsymbol{\sigma}| = 1
\end{array}
$$

Example (162)

Example 51

$$
\begin{array}{ll}
(\mathrm{M}[\![\neg(\mathrm{Account} < 0)]\!](\boldsymbol{\sigma}) = \mathrm{tt}) & \mathrm{iff} \\
\mathrm{not}\ (\mathrm{M}[\![\mathrm{Account} < 0]\!](\boldsymbol{\sigma}) = \mathrm{tt}) & \mathrm{iff} \\
\mathrm{not}\ (\mathrm{E}[\![\mathrm{Account}]\!](\boldsymbol{\sigma}) < \mathrm{E}[\![0]\!](\boldsymbol{\sigma})) & \mathrm{iff} \\
\mathrm{not}\ (\boldsymbol{\sigma}_0(\mathrm{Account}) < 0) &
\end{array}
$$

Example 52

$$
\begin{array}{ll}
(\mathrm{M}[\![\mathrm{Account} = 50 \wedge \mathrm{In} \geq 0]\!](\boldsymbol{\sigma}) = \mathrm{tt}) & \mathrm{iff} \\
(\mathrm{M}[\![\mathrm{Account} = 50]\!](\boldsymbol{\sigma}) = \mathrm{tt})\ \mathrm{and}\ (\mathrm{M}[\![\mathrm{In} \geq 0]\!](\boldsymbol{\sigma}) = \mathrm{tt}) & \mathrm{iff} \\
(\mathrm{E}[\![\mathrm{Account}]\!](\boldsymbol{\sigma}) = \mathrm{E}[\![50]\!](\boldsymbol{\sigma}))\ \mathrm{and}\ (\mathrm{E}[\![\mathrm{In}]\!](\boldsymbol{\sigma}) \geq \mathrm{E}[\![0]\!](\boldsymbol{\sigma})) & \mathrm{iff} \\
\boldsymbol{\sigma}_0(\mathrm{Account}) = 50\ \mathrm{and}\ \boldsymbol{\sigma}_0(\mathrm{In}) \geq 0 &
\end{array}
$$

Semantics of formulae (163)

Let $\boldsymbol{\sigma} \sim_V \boldsymbol{\sigma}'$ denote that the intervals $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}'$ are identical with the possible exception of the mapping for the variable V.

Example 53

Let $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}'$ be intervals.

- Let $\boldsymbol{\sigma}_0(\mathrm{Cash}) = 0$ and $\boldsymbol{\sigma}_0(\mathrm{In}) = 2$.

  Let $\boldsymbol{\sigma}_1(\mathrm{Cash}) = 1$ and $\boldsymbol{\sigma}_1(\mathrm{In}) = 4$.

  Let $\boldsymbol{\sigma}_2(\mathrm{Cash}) = 1$ and $\boldsymbol{\sigma}_2(\mathrm{In}) = 3$.
- Let $\boldsymbol{\sigma}'_0(\mathrm{Cash}) = 0$ and $\boldsymbol{\sigma}'_0(\mathrm{In}) = 3$.

  Let $\boldsymbol{\sigma}'_1(\mathrm{Cash}) = 1$ and $\boldsymbol{\sigma}'_0(\mathrm{In}) = 2$.

  Let $\boldsymbol{\sigma}'_2(\mathrm{Cash}) = 1$ and $\boldsymbol{\sigma}'_2(\mathrm{In}) = 3$.

Then $\boldsymbol{\sigma} \sim_{\mathrm{In}} \boldsymbol{\sigma}'$.

Semantics of formulae (164)

The semantics of $\forall V \bullet f$ is defined in terms of $\boldsymbol{\sigma} \sim_V \boldsymbol{\sigma}'$

$$\mathrm{M}[\![\forall V \bullet f]\!](\boldsymbol{\sigma}) = \mathrm{tt} \quad \mathrm{iff} \quad (\mathrm{for\ all}\ \boldsymbol{\sigma}'\ \mathrm{s.t.}\ \boldsymbol{\sigma} \sim_V \boldsymbol{\sigma}', \mathrm{M}[\![f]\!](\boldsymbol{\sigma}') = \mathrm{tt})$$

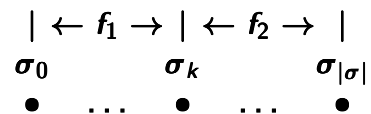
Example 54

$$
\begin{array}{ll}
\mathrm{M}[\![\forall \mathrm{In} \bullet \mathrm{In} \geq 0]\!](\boldsymbol{\sigma}) = \mathrm{tt} & \mathrm{iff} \\
(\mathrm{for\ all}\ \boldsymbol{\sigma}'\ \mathrm{s.t.}\ \boldsymbol{\sigma} \sim_{\mathrm{In}} \boldsymbol{\sigma}', \mathrm{M}[\![\mathrm{In} \geq 0]\!](\boldsymbol{\sigma}) = \mathrm{tt}) & \mathrm{iff} \\
(\mathrm{for\ all}\ \boldsymbol{\sigma}'\ \mathrm{s.t.}\ \boldsymbol{\sigma} \sim_{\mathrm{In}} \boldsymbol{\sigma}', \boldsymbol{\sigma}'_0(\mathrm{In}) \geq 0) &
\end{array}
$$

Semantics of formulae (165)

The semantics of 'chop' is as follows $M[\![f_1 ; f_2]\!](\sigma) =$ tt iff

(exists k, s.t. $M[\![f_1]\!](\sigma_0 \dots \sigma_k) =$ tt and $M[\![f_2]\!](\sigma_k \dots \sigma_{|\sigma|}) =$ tt)

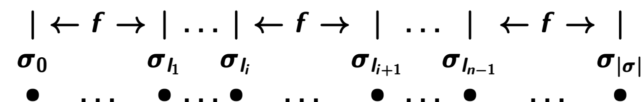


Interval σ is a fusion of two intervals $\sigma_0 \dots \sigma_k$ (satisfies f_1) and $\sigma_k \dots \sigma_{|\sigma|}$ (satisfies f_2). State σ_k is shared by both.

Semantics of formulae (166)

The semantics of 'chopstar' is as follows $M[\![f^*]\!](\sigma) =$ tt iff

(exist $l_0, \dots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
for all $0 \leq i < n, l_i \leq l_{i+1}$ and $M[\![f]\!](\sigma_{l_i} \dots \sigma_{l_{i+1}}) =$ tt)



Finite interval σ is the fusion of a finite number of finite sub-intervals each satisfying f.

Satisfiable and valid (167)

- A first order ITL formula f is satisfiable if and only if there exists an interval σ such that $M[\![f]\!](\sigma) =$ tt.
- A first order ITL formula f is valid if and only if for all intervals σ, $M[\![f]\!](\sigma) =$ tt.

Example 55

- $\Box(0 < 1)$ is a valid formula.
- Formula skip $\wedge A = 0$ is satisfiable because $M[\![\text{skip} \wedge A = 0]\!](\sigma) =$ tt where interval σ is $\sigma_0\sigma_1$ and $\sigma_0(A) = 0$ and $\sigma_1(A) = 10$.

Summary (168)

- A first order ITL formula contains predicates, propositional variables, Boolean operators, temporal operators and quantifiers.
- The semantics (meaning) of a first order ITL formula is defined with respect to an interval (non-empty sequence of states).
- One can define the satisfiability and validity of an ITL formula.

Exercises – 1

Exercise 26

Give for each of the following intervals the corresponding Interval Temporal Logic formula

1. $\bullet$ $\bullet$ $\bullet$
   $\neg P, \neg Q$ $\quad P, Q$ $\quad P, Q$
2. $\bullet$ $\bullet$ $\bullet$ $\bullet$
   P $\quad P$ $\quad \neg P$ $\quad \neg P$
3. $\bullet$ $\bullet$ $\bullet$ $\bullet$ $\bullet$
   $P, \neg Q$ $\quad \neg P, Q$ $\quad P, \neg Q$ $\quad \neg P, Q$ $\quad P, \neg Q$
4. $\bullet$ $\bullet$ $\bullet$ $\bullet$
   $I = 1$ $\quad I = 3$ $\quad I = 9$ $\quad I = 27$

Exercises – 2

Exercise 27

Give an English description of the interval that correspond to each of the following Interval Temporal Logic formulae

1. $(\text{halt } Q) \wedge \text{skip}^4$
2. $(P \wedge \text{skip}^2 \wedge \text{fin } (Q))\ ;\ (\neg P \wedge \text{skip} \wedge \text{fin } (Q))$
3. $((\Box\, Q)\ ;\ (\text{skip}^3 \wedge \text{halt } P)) \wedge \text{skip}^5$
4. $(\text{skip}^3)^*$

Exercises – 3

Exercise 28

Give the informal semantics (picture) of following formulae:

- $A = 0 \wedge \text{skip} \wedge A := A + 1$
- $\text{len}(6)$
- $A = 0 \wedge \text{len}(2) \wedge A \leftarrow A + 1$
- $\text{len}(4) \wedge A = 3 \wedge A \text{ gets } A + 1$
- $\text{len}(3) \wedge A = 2 \wedge \text{stable } A$

Exercises – 4

Exercise 29

Given informal specification

- *the system's behaviour consists of only two states,*
- *in the initial state the variable* Account *is equal to* **50** *and*
- *in the next state it is increased by* **100**.

Give the corresponding ITL formula.

Exercises – 5

Exercise 30

Given the following informal specification

> *An interval where in the initial state the value of variable* $\boldsymbol{I}$ *is* $\mathbf{6}$ *and the value of variable* $\boldsymbol{Q}$ *is* false. *The value of* $\boldsymbol{I}$ *in each next state is decremented by* $\mathbf{1}$ *until the value of* $\boldsymbol{I}$ *reaches* $\mathbf{0}$ *whereas the value of* $\boldsymbol{Q}$ *in each next state is the negation of the value of* $\boldsymbol{Q}$ *in the current state.*

1. *Give its corresponding Interval Temporal Logic formula.*
2. *Give its corresponding semantics in pictorial form.*

Exercises – 6

Exercise 31

Determine which of the following formula is satisfiable or valid. Explain why.

- $\Diamond((\boldsymbol{A} + \boldsymbol{B}) = (\boldsymbol{B} + \boldsymbol{A}))$
- $\exists \boldsymbol{A} \bullet (\text{skip} \,;\, (\boldsymbol{A} = \mathbf{0}))$
- $\exists \boldsymbol{A} \bullet (\boldsymbol{A} = \mathbf{0} \wedge \boldsymbol{A} \leftarrow \boldsymbol{A} + \mathbf{1} \wedge \text{len}(\mathbf{5}))$
- $\forall \boldsymbol{A} \bullet (\text{IsOdd}(\boldsymbol{A}) \,;\, \text{skip})$

Part VI

Tempura

Overview

- Introduction
- Justification
- Tempura Syntax
- Locations
- Expressions
- Statements
- Liquid Tank Controller
- Robot Control System
- Alternating Bit Protocol
- Summary
- Exercises

Introduction (177)

- Programming language based on Interval Temporal Logic.
- Provides a way of directly executing suitable Interval Temporal Logic specifications of digital circuits, parallel programs and other dynamic systems.
- Since every Tempura statement is also a temporal formula, the entire Interval Temporal Logic formalism can be used to reason about Tempura programs.
- Tempura has the two seemingly contradictory properties of being a logic programming language and having imperative constructs such as assignment statements.

Justification – 1 (178)

- Executable specifications allow demonstrating the behaviour of a system before it is actually implemented. This has three positive consequences for system development:
  - executable components are much earlier available than in traditional life cycle therefore validation errors can be corrected immediately without incurring costly redevelopment.
  - requirements that are initially unclear can be clarified and completed by hands-on experience with the executable specification.
  - execution of the specification supplements inspection and reasoning as means for validation. This is especially important for the validation of non-functional behaviour.

Justification – 2 (179)

- Declarative languages, especially logic languages, combine high expressiveness with executability. They allow writing both property-oriented and model-oriented executable specifications on the required level of abstraction.
- Executable specifications are constructive, i.e. they do not only demand the existence of a solution, they actually construct it.
- Non-executable specifications which are constructive can be transformed into executable ones on almost the same level of abstraction. The resulting executable specifications are often based on search. It is not necessary to introduce new algorithms to achieve executability.

Justification – 3 (180)

- Executable specifications do not necessarily constrain the choice of possible implementations because only minimal design and implementation decisions are necessary to get executability. In addition, these decisions are revisable.
- Verification of an implementation against the specification becomes superfluous if one uses the transformational approach.

Example (181)

Consider the following ITL formula

Example 56

$M = 4 \wedge N = 1 \wedge \text{halt}\,(M = 0) \wedge M \text{ gets } M - 1 \wedge N \text{ gets } 2 * N$

Interval:
State 0: M=4 N=1
State 1: M=3 N=2
State 2: M=2 N=4
State 3: M=1 N=8
State 4: M=0 N=16

Tempura interpreter – 1 (182)

Tempura interpreter:

- Given a executable ITL (Tempura) formula
- Generate a satisfying interval

A formula is executable if

- it is deterministic,
- the length of the satisfying interval is known.
- the values of the variables (of the formula) are known throughout the satisfying interval.

Tempura interpreter – 2 (183)

Tempura interpreter:

takes a Tempura formula and constructs the satisfying sequence of states, i.e. interval, using the following rewrite technique:

$f \equiv \text{present_state} \wedge ⓦ\text{what_remains}$

Example 57

$$\bigcirc f \equiv \text{more} \wedge ⓦ f$$
$$\Box f \equiv f \wedge ⓦ\Box f$$

Tempura interpreter – 3 (184)

Example 58

Some examples of executable formulae:

$\text{skip} \wedge I = 0 \wedge \bigcirc(I = 1)$
$\text{len}(3) \wedge \Box(I = 2)$
$\text{len}(4) \wedge I = 0 \wedge I \text{ gets } I + 1$

Example 59

Some examples of non-executable formulae:

$\text{empty} \wedge (I = 0 \vee I = 1)$
$I = 0 \,;\, I = 1$
$\text{len}(4) \wedge I \text{ gets } I + 1$

Tempura Syntax (185)

Three main categories:

- Locations: place where values are stored and examined, e.g., variables $M, N, \ldots,$ **next** $M,$ **next** $N, \ldots$.
- Expressions: integer, lambda, Boolean, list, string and float.
- Statements: subset of ITL plus system statements.

Variables (186)

- Static variables start with a lower case letter. Static variables are constant on any given interval.
- State variables start with an uppercase letter. State variables change in value from state to state.

  The value of a state variable on an interval is defined to be its value on the first state. Thus, I=0 means I has initial value zero.
- If V is a variable then so is **next** V.

Variables (187)

Variables and expressions may be of any of the following types:

integer:	0, 42, -97, etc.
Boolean:	true, false.
list:	Lists may be of fixed or variable length.
string:	String constants are of the form "abc".
float:	Float constants are of the form 0.123 or $0.123e+10$.
lambda:	This is the type of predicates and functions.

The function type(X) returns a string representing the type of expression X. For example, type(9)="integer".

Integer Expressions (188)

$$\begin{aligned} ae ::= \; & z \mid \textbf{random} \mid \textbf{Random} \mid |le| \mid a \mid \textbf{next}\ a \mid A \mid \textbf{next}\ A \mid \\ & ae_1 + ae_2 \mid ae_1 - ae_2 \mid ae_1 * ae_2 \mid \\ & ae_1\ \textbf{div}\ ae_2 \mid ae_1\ \textbf{mod}\ ae_2 \mid ae_1 ** ae_2 \mid \\ & \textbf{if}\ be\ \textbf{then}\ ae_1\ \textbf{else}\ ae_2 \mid f(e_1, \ldots, e_n) \end{aligned}$$

random is an integer random number (static within a state).
Random is an integer random number (can change within a state).

Example 60

$1 + \textbf{next}\ a$
$A + \textbf{next}\ B$
$\textbf{if}\ ((\textbf{next}\ A) > B)\ \textbf{then}\ C + \textbf{next}\ B\ \textbf{else}\ (\textbf{next}\ C) + D$

Boolean Expressions (189)

be ::= **true** | **false** | a | **next** a | A | **next** A |
be$_1$ **or** be$_2$ | be$_1$ **and** be$_2$ | $\sim$ be |
empty | **more** | e$_1$ $=$ e$_2$ | e$_1$ $\sim=$ e$_2$ |
ae$_1$<ae$_2$ | ae$_1$<=ae$_2$ | ae$_1$>ae$_2$ | ae$_1$>=ae$_2$ |
if be **then** be$_1$ **else** be$_2$ | f(e$_1$, . . . ,e$_n$) |
exists i <ae:{f(i,e$_1$, . . . ,e$_n$)} |
forall i <ae:{f(i,e$_1$, . . . ,e$_n$)}

Example 61

empty or (next $A > 0$**)**
exists $i < |L| : \{L[i] > 5\}$

List Expressions (190)

le ::= a | **next** a | A | **next** A |
le$_1$+le$_2$ | le[ae] | [e$_1$, . . . ,e$_n$] | le[ae$_1$..ae$_2$]

To access an element of a list, the notation $L[i]$ should be used. Indexing begins at zero. Thus, for example, if $L = [1, 2, 3, 4]$ then $L[2] = 3$.

Sublist $L[i..j]$ denotes a sublist of L from i to $j - 1$. For example, **list** $(L, 5)$ **and** $L[1..3] = [1, 2]$ assigns the values 1 and 2 to the second and third elements of the list.

Example 62

$L1$ + **next** $L2$
$L[2..5] = [L[2], L[3], L[4]]$
List $(L, 5)$

List Expressions (191)

Declaration:

- **list** (L, n): defines L to be a list of fixed size n.
- **List** (L, n): defines L to be a list of variable size up to maximum of n.
- **list** (L, n) and **List** (L, n) apply only on the current state.
- **stable (struct** (L)**)** keeps the list structure of L stable over an interval.

Example 63

List $(L, 5)$ **and stable (struct** (L)**)**

String Expressions (192)

se ::= a | **next** a | A | **next** A |
se$_1$+se$_2$ | se[ae] | se[ae$_1$..ae$_2$] |
"$s_1 \dots s_n$" | [s_1, . . . ,s_n]

String constants are enclosed in double quotes, e.g. "this is a string". Most of the C escapes may be used. For example `\n` introduces a newline character, and `\"` introduces a double quote. The character `\` itself must be escaped as `\\`.

Two strings may be appended with the + operator: "abc"+"def"="abcdef".

Example 64

$L1$ + **next** $L2$
$L[2..5] = [L[2], L[3], L[4]]$
L = "*abc*"

Float Expressions (193)

fe ::= $\$d.d\$$ | $\$d.d$ e$\pm$ $d\$$ | **frandom** | **fRandom** | a | **next** a |
A | **next** A | fe_1+fe_2 | fe_1-ae_2 | fe_1*fe_2 |
fe_1 **div** fe_2 | fe_1 **mod** fe_2 | fe_1**fe_2 |
fe_1/fe_2 | **sqrt** (fe) | **ceil** (fe) | **floor** (fe) | **exp** (fe) |
log (fe) | **log10** (fe) | **sin** (fe) | **cos** (fe) | **tan** (fe) |
asin (fe) | **acos** (fe) | **atan** (fe) | **atan2** (fe_1,fe_2) |
sinh (fe) | **cosh** (fe) | **tanh** (fe) | **fabs** (fe) | **itof** (ae) |
if be **then** fe_1 **else** fe_2 | $f(e_1,\ldots,e_n)$

The function **itof** returns the float corresponding to an integer. **floor** and **ceil** return respectively the largest integer not greater than, smallest integer not less than. **frandom** is a float random number in [0.0,1.0) (static within a state). **fRandom** is a float random number in [0.0,1.0) (can change within a state).

Example 65

$A = \cos(\$0.123\$)$
next $B = \exp(\$0.123e+10\$)$

Lambda Expressions (194)

ge ::= **lambda** $(V_1,\ldots,V_n)$:{e}

The command **define** $F(Y) = \{Y+1\}$ to define a function F is the same as the assignment $F =$ **lambda** $(Y) : \{Y+1\}$.

Example 66

$F =$ **lambda** $(Y, X) : \{X+Y\}$
$G =$ **lambda** $(Y) : \{$**lambda** $(X) : \{X+Y\}\}$

Statements (195)

- Simple statements: skip, empty, more, length, assignments, list statements.
- Compound statements: choice, loop, sequential, parallel, always, sometimes, functions, predicates.

Simple (196)

empty | **more** | **skip** | **len** (ae)

The len construct can be used to define any interval length. The term len(n) defines an interval of length n (containing n+1 states). Thus,

len (0) = **empty**
len (1) = **skip**

Example 67

$A = 0$ **and empty**
$A = 0$ **and more**
$A = 0$ **and skip**

Assignments (197)

- simple: $a = e$, $A = e$
- unit: **(next** A**)** $= e$, $A := e$
- gets: A **gets** e
- temporal: $A \Leftarrow e$
- stable: **stable** (A)

Example 68

$A = 5 + B$
$A := A + 1$
A **gets** $B + 1$
$A \Leftarrow B + 1$
stable (B)

Choice (198)

if be **then** st_1 **else** st_2 | **if** be **then** st_1 | be **implies** st_1

The form **if** ... **then** ... is the same as **if** ... **then** ... **else true** and is the same as ... **implies** ...

Example 69

if $A > 0$ **then** $B := B + 4$ **else** $B := B + 2$
if $A > 0$ **then** $B := A + 4$

Loop (199)

while be **do** st | **repeat** st **until** be |
for a <ae **do** st | **chopstar** (st) |
for a **in** le **do** st | **for** a **in** se **do** st | **for** ae **times do** st

The **while**, **repeat**, **chopstar** and **for** statements define sequential loops in the usual way.

Note that the control variable of a **for** loop should be static.

Example 70

while $A > 0$ **do** $A := A - 1$
repeat $B := B + 1$ **until** $B = 5$
for $i < n$ **do** $L[i] := L[i] + 1$
chopstar $(A := A + 1)$
$I = [0, 1, 2]$ **and for** e **in** I **do** {**skip and output** (e)}

Sequential (200)

st_1;st_2

The **;** operator stands for sequential composition. The formula before the **;** must define an interval length. For example,

$\{I = 0$ **and** I **gets** $I + 1\}$; $\{$**len** (4) **and** I **gets** $I + 1\}$ is not executable, since it is satisfied by a unbounded number of behaviours.

Example 71

$\{A = 0$ **and empty**$\}$; $\{B = 0\}$
$\{A := A + 1$ **and skip**$\}$; $\{B := B + 1\}$

Parallel (201)

st$_1$ **and** st$_2$

The logical conjunction operator **and** may be used to compose programs in parallel. For example, the conjunction

len (5) and always $\{I = 0\}$ **and always** $\{J = 1\}$

causes the terms **len (5)**, **always** $\{I = 0\}$, and **always** $\{J = 1\}$ to be executed concurrently. That is, it defines a new interval of length 5 on which I is always 0 and J is always 1.

Example 72

$A = 0$ **and** $B = 1$ **and** $A := A + 1$ **and** $B := B + 1$ **and** skip
$\{A = 0$ **and empty**$\}$; $\{B = 0$ **and empty**$\}$
$\{A := A + 1$ **and stable** $(B)\}$; $\{B := B + 1$ **and stable** $(A)\}$

Next (202)

next st

The operator **next** is the ○ operator. The term **next** ... causes ... to be executed on the next state. There **must** be a next state. Thus,

next $\{I = 1\}$

means that the variable I has the value 1 on the next state. It fails if the interval is empty.

Example 73

next $(A = 0)$
$B = 0$ **and next** $(A = B + 1)$

Always (203)

always st

The operator **always** is the □ operator. The term **always** ... causes ...to be executed on every state. Thus,

always $\{I = 1\}$

means that the variable I always has the value 1.

Example 74

always $(A = 1)$
always $(A := A + 1)$
always (**if more then** $A := A + 1$)

Sometimes (204)

sometimes st

The operator **sometimes** is ◇ operator. The term **sometimes** ... causes the interpreter to check that ... is **true** at some time in a program. Thus,

$I = 0$ **and** I **gets** $I + 1$ **and halt** $(I = 10)$ **and sometimes** $\{I = 5\}$

will succeed.

Example 75

$I = 0$ **and** I **gets** $I + 1$ **and len** (2) **and sometimes** $(I = 5)$
$I = 0$ **and** I **gets** $I + 1$ **and len** (7) **and sometimes** $(I = 2)$

Halt (205)

halt (be)

The **halt** operator defines the termination condition for an interval. For example, the program:

$\boldsymbol{I = 0}$ **and** $\boldsymbol{I}$ **gets** $\boldsymbol{I + 1}$ **and halt** $\boldsymbol{(I = 5)}$

defines an interval on which I is incremented state-by-state until it equals 5. The interval length must therefore be 5.

Example 76

$\boldsymbol{I = 5}$ **and** $\boldsymbol{I}$ **gets** $\boldsymbol{I + 1}$ **and halt** $\boldsymbol{(I = 5)}$

Fin (206)

fin (be)

The term **fin** ... means that ... must be true on the last state of the interval. Thus, the program

len (5) **and keep** $\{I = 0\}$ **and fin** $\{I = 1\}$

defines an interval of length 5 for which I is 1 on the last state, and 0 on every other state.

Example 77

$J = 0$ **and stable** (J) **and**
$I = 0$ **and** I **gets** $I + 1$ **and len** (3) **and fin** $(J = 0)$

Exists (207)

exists A : st | **exists** a : st | **exists** $V_1, \ldots, V_n$: st

Existential quantification is denoted by the **exists** operator and semantically, it corresponds to the introduction of local variables. For example, in the program

exists I, J : $\{I = 0$ **and** $J = 1$ **and** ... **and**
exists I : $\{I = J$ **and** ...$\}$
$\}$

there are two instances of the variable I. The outer one has initial value 0, the inner one has initial value 1.

Example 78

exists I, J : $\{J = 0$ **and stable** (J) **and**
$I = 0$ **and** I **gets** $I + 1$ **and**
len (3) **and fin** $(J = 0)\}$

Forall (208)

forall a <ae : st

Universal quantification is denoted by the **forall** operator. Only the bounded form **forall** $i < n : \{\ldots\}$ is permitted. Semantically, universal quantification corresponds to indexed concurrency. The program

forall $i < n : \{p(i)\}$

is equivalent to $p(0)$ **and** ... **and** $p(n - 1)$.

Example 79

forall $i < |L| : \{L[i] = 0\}$
forall $i < |L| : \{L[i] := L[i] + 1\}$

Functions

define $F(V_1, \ldots, V_n) = \{e\}$.

The **define** construct can be use to define functions. The parameters of a function call are passed via ‘call by reference’.

Example 80

define $max(I, J) = \{$**if** $I >= J$ **then** I **else** $J\}$.
$A := A + max(C, D)$

Procedures

define $P(V_1, \ldots, V_n) = \{\text{st}\}$.

The **define** construct is used to define procedures. The parameters of a procedure call are passed via ‘call by reference’.

Example 81

```
define hanoi(n) = exists L,C,R : {
    list(L,n) and forall i<n : L[i]=i and
    C=[] and R=[] and move(n,L,C,R) and
    always format("L=%10t C=%10t R=%10t\n",L,C,R)
}.
/* Move n discs from peg A to peg B */
define move(n,A,B,C) = {
    if n=0 then empty
    else {
        move(n-1,A,C,B);
        {skip and A:=A[1..|A|] and B:=A[0..1]+B and C:=C};
        move(n-1,C,B,A)
    }
}.
```

Input/Output

input (V) | **output** (e) | **format** ("*string*",$e_1, \ldots, e_n$)

The **input**, **output** and **format** constructs are for input and output. The “string” in **format** is like C.

Example 82

exists I, J : {**input** (I) **and** $J = 0$ **and**
  always output (J) **and halt** $(I = 0)$ **and**
  $(I$ **gets** $I - 1)$ **and** $(J$ **gets** $J + I)$
}.

System commands

load "file". | **run** st.

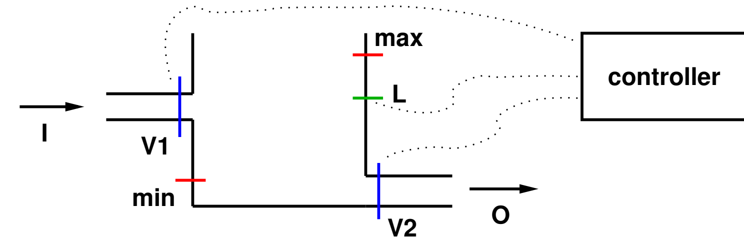
The **load** command is used to load a file containing a program. The **run** command is used to run a program.

Liquid Tank Controller: Intro (213)

Simple control system maintaining the level of liquid in a tank.

- A tank has an inlet pipe and an outlet pipe. The flow in each pipe is controlled by a simple valve, which can be either open or closed.
- A controller unit controls the valves on the inlet and outlet pipes to maintain the level of the liquid in the tank within some specified limits.

Liquid Tank Controller V1 – 1 (214)



- V1 : valve on the inlet pipe; open/closed
- V2 : valve on the outlet pipe; open/closed
- I : rate of flow of liquid in the inlet pipe
- O : rate of flow of liquid in the outlet pipe
- L : the level of liquid in the tank
- min, max : the given minimum and maximum levels

Liquid Tank Controller V1 – 2 (215)

```
define min = 5.
define max = 10.
define closed = 0.
define open = 1.

define example() = {
  exists V1, V2, I, O, L : {
   initialise(V1, V2, L) and
   keep {
    pipe(V1, I, 0, 3) and
    pipe(V2, O, 1, 1) and
    tank(I, O, L) and
    controller(V1, V2, L) and
    format ("V1=%d, V2=%d, I=%d, O=%d, L=%d\n",
                V1, V2, I, O, L)
   } and len (50)
  }
}.
```

Liquid Tank Controller V1 – 3 (216)

```
/* Initial state. Note valves closed imply that the flows are also zero. */
define initialise(V1, V2, L) = {
  V1 = closed and
  V2 = closed and
  L = 0
}.

/* Defines a flow rate F between minf and maxf, or zero if V is closed. */
define pipe(V, F, minf, maxf) = {
  if (V = open) then
    F = (Random mod (maxf − minf + 1)) + minf
  else F = 0
}.
```

Liquid Tank Controller V1 – 4


```
/* The level of liquid in a unit interval. */
define tank(I, O, L) = {
  L := L + I − O
}.

/* The state of each valve in a unit interval */
define controller(V1, V2, L) = {
  if L < max then {V1 := open} else {V1 := closed} and
  if L > min then {V2 := open} else {V2 := closed}
}.
```

Liquid Tank Controller V1 – 5

Run of the controller:

```
State  0: V1=0, V2=0, I=0, O=0, L=0
...
State 13: V1=1, V2=1, I=2, O=1, L=10
State 14: V1=0, V2=1, I=0, O=1, L=11
State 15: V1=0, V2=1, I=0, O=1, L=10
...
State 27: V1=1, V2=1, I=0, O=1, L=7
State 28: V1=1, V2=1, I=2, O=1, L=6
State 29: V1=1, V2=1, I=0, O=1, L=7
...
State 42: V1=1, V2=1, I=2, O=1, L=10
State 43: V1=0, V2=1, I=0, O=1, L=11
State 44: V1=0, V2=1, I=0, O=1, L=10
...
```

So L does not stay between 5 and 10!

Liquid Tank Controller V2 – 1

- V1 : valve on the inlet pipe; open/closed
- V2 : valve on the outlet pipe; open/closed
- I1 : rate of flow of liquid willing to flow into the inlet pipe
- I : rate of flow of liquid in the inlet pipe
- O1 : rate of flow of liquid willing to flow into the outlet pipe
- O : rate of flow of liquid in the outlet pipe
- L : the level of liquid in the tank
- min, max : the given minimum and maximum levels

Liquid Tank Controller V2 – 2


```
define min = 5.
define max = 10.
define closed = 0.
define open = 1.
define example() = {
  exists V1, V2, I1, I, O1, O, L : {
   initialise(L) and
   keep {
    pipe(V1, I1, I, 0, 3) and
    pipe(V2, O1, O, 1, 1) and
    tank(I, O, L) and
    controller(V1, V2, L, I1, O1) and
    format ("V1=%d, I1=%d, I=%d,
             V2=%d, O1=%d, O=%d, L=%d\n",
              V1, I1, I, V2, O1, O, L)
   } and len (50)
  }
}.
```

Liquid Tank Controller V2 – 3


```
/* Initial state. Note initial tank level is min. */
define initialise(L) = {
  L = min
}.

/* Defines a flow rate F between minf and maxf,
 Fw represents the willing flow while
 Fa represents the actually flow which is either the willing flow if
 the valve is open and zero when valve V is closed. */
define pipe(V, Fw, Fa, minf, maxf) = {
  Fw = (Random mod (maxf − minf + 1)) + minf and
  if (V = open) then Fa = Fw else Fa = 0
}.
```

Liquid Tank Controller V2 – 4


```
/* The level of liquid in a unit interval. */
define tank(I, O, L) = {
  L := L + I − O
}.

/* The state of each valve in a unit interval */
define controller(V1, V2, I1, O1, L) = {
  if L + I1 − O1 <= max then {V1 = open} else {V1 = closed} and
  if L + I1 − O1 >= min then {V2 = open} else {V2 = closed}
}.
```

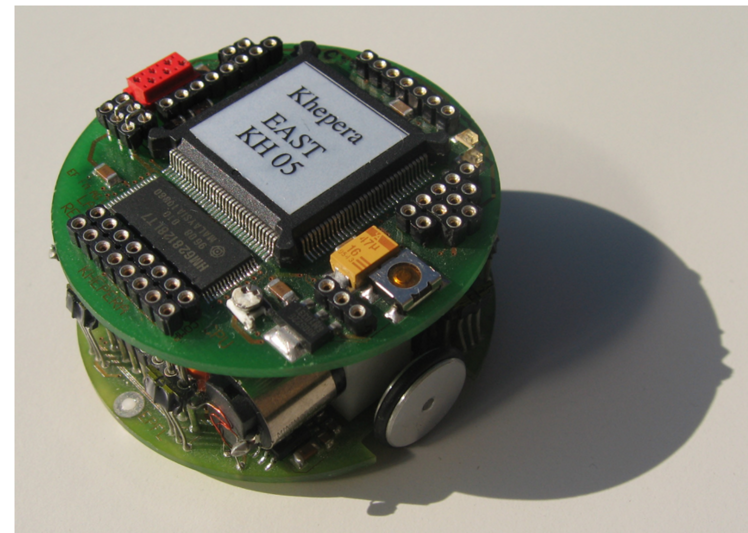
Liquid Tank Controller V2 – 5

Run of the controller:

```
State   0: V1=1, I1=2, I=2, V2=1, O1=1, O=1, L=5
State   1: V1=1, I1=2, I=2, V2=1, O1=1, O=1, L=6
State   2: V1=1, I1=2, I=2, V2=1, O1=1, O=1, L=7
State   3: V1=1, I1=3, I=3, V2=1, O1=1, O=1, L=8
State   4: V1=1, I1=0, I=0, V2=1, O1=1, O=1, L=10
State   5: V1=1, I1=1, I=1, V2=1, O1=1, O=1, L=9
...
...
State  44: V1=0, I1=3, I=0, V2=1, O1=1, O=1, L=10
State  45: V1=0, I1=3, I=0, V2=1, O1=1, O=1, L=9
State  46: V1=1, I1=0, I=0, V2=1, O1=1, O=1, L=8
State  47: V1=1, I1=1, I=1, V2=1, O1=1, O=1, L=7
State  48: V1=1, I1=2, I=2, V2=1, O1=1, O=1, L=7
State  49: V1=1, I1=1, I=1, V2=1, O1=1, O=1, L=8
```

Now L does stay between 5 and 10!

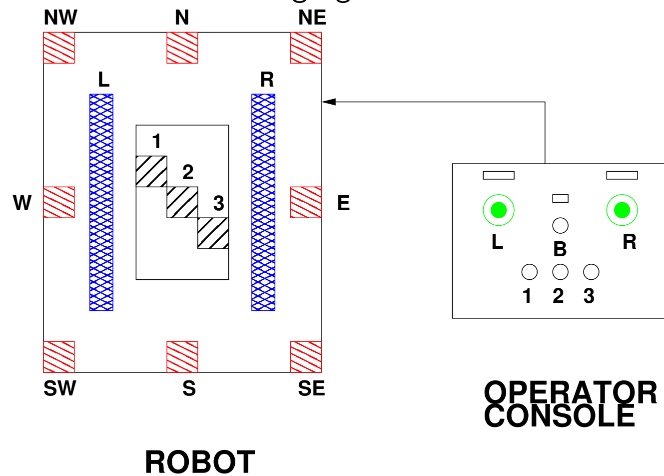
Robot Control System: Intro – 1



Wikipedia

Robot Control System: Intro – 2 (225)

A tele-operated robot is a tracked device with vehicle schematic as shown in the following figure.



Robot Control System: Intro – 3 (226)

- The actual vehicle is driven by two motors, left and right. Both of these motors can move forwards and in reverse. The vehicle is steered by moving one motor faster than the other.
- From a control point of view, commands are issued to the motors via a operator joystick which issues integers values in the range $\mathbf{0 \ldots 127}$ for forward motion (127 max. speed) and $\mathbf{0 \ldots -128}$ for reverse motion. It is possible to drive only one motor at a time, in such a case the robot will turn. The speed of the motors is directly proportional to the value written to them.

Robot Control System: Intro – 4 (227)

- The robot is equipped with 8 infra red sensors. These return an integer value in the range $\mathbf{0 \ldots 255}$ depending on whether an obstacle is present or not. $\mathbf{0}$ indicates no obstacle, $\mathbf{255}$ indicates obstacle very near. We normally operate with a threshold of around $\mathbf{100}$, above which we take notice of the sensor readings, i.e., an obstacle is of interest. At this point reactive control takes over from the manual control by moving the vehicle away from the obstacle until the $\mathbf{100}$ threshold is not set. The sensor positions are as follows: N, NE, E, SE, S, SW, W and NW, covering the body of the robot and shown in the figure.
- The specification of the robot control system consists of 3 parts: **Motor control**, **Infra-red control** and **Operator control**.

Robot Control System: ITL – 1 (228)

- **Motor control**: If the sensor detects an object then the control system takes over control otherwise if the operator requests a new movement then action this.

  Let Lic and Ric denote respectively the left and right motor commands issued by the infra-red control. Let Iractive denote the presence/absence of an object. Let Loc and Roc denote the left and right motor command issued by the operator and let Opactive denote an active operator request.

  Let move$(\boldsymbol{l}, \boldsymbol{r})$ denote the sending of left $\boldsymbol{l}$ and right $\boldsymbol{r}$ motor commands to the two motors.

Robot Control System: ITL – 2 (229)

- **Motor control**:
  MCS $\triangleq$
  $$(\ (\text{Iractive} \wedge \text{move}(\text{Lic}, \text{Ric})) \vee (\text{Opactive} \wedge \text{move}(\text{Loc}, \text{Roc})) \vee (\neg\text{Iractive} \wedge \neg\text{Opactive}) \ )^*$$
  What happens when both Iractive and Opactive are true? Both move(Lic, Ric)) and move(Loc, Roc) commands are issued.
- Need to resolve which command has priority. If Infra-red has priority we have:
  MCS $\triangleq$
  $$(\ (\text{Iractive} \wedge \text{move}(\text{Lic}, \text{Ric})) \vee (\text{Opactive} \wedge \neg\text{Iractive} \wedge \text{move}(\text{Loc}, \text{Roc})) \vee (\neg\text{Iractive} \wedge \neg\text{Opactive}) \ )^*$$
  We have strengthened the guard of the Operator with the negation of the Infra-red guard.

Robot Control System: ITL – 3 (230)

- **Infra-red control**: read the sensors and for each sensor that is greater than the threshold of **100** adjust the motor commands accordingly. For example if the north sensor detects an object we should move in the south direction as an avoidance strategy.

  Let Ircounts(i) denote the sensor i (N: i=0, NE: i=1, E: i=2, SE: i=3, S: i=4, SW: i=5, W: i=6, NW: i=7). Let mvl(i) and mvr(i) denote respectively the left and right steering commands corresponding to sensor i.

	0	1	2	3	4	5	6	7
mvl	*b*	*s*	*b*	*s*	*f*	*f*	*f*	*b*
mvr	*b*	*b*	*f*	*f*	*f*	*s*	*b*	*s*
	↓	↙	←	↖	↑	↗	→	↘

ICS $\triangleq$
$$(\ \text{Iractive} = (\textstyle\bigvee_i \text{Ircounts}(i) > 100) \wedge \text{Lic} = (\sum_i : \text{Ircounts}(i) > 100 : \text{mvl}(i)) \wedge \text{Ric} = (\sum_i : \text{Ircounts}(i) > 100 : \text{mvr}(i)) \ )^*$$

Robot Control System: ITL – 4 (231)

- **Operator control**: if the operator requests some changes then process them.

  Let Loc and Roc denote respectively the left and right steering commands received from the operator. Let Lloc and Lroc denote respectively the last left and last right steering commands received from the operator.

  OCS $\triangleq$
  $$\exists \text{Lloc}, \text{Lroc} \bullet (\ \text{Lloc} = 0 \wedge \text{Lroc} = 0 \wedge (\text{Opactive} = (\text{Loc} \neq \text{Lloc} \vee \text{Roc} \neq \text{Lroc}) \wedge \bigcirc\text{Lloc} = \text{Loc} \wedge \bigcirc\text{Lroc} = \text{Roc} \ )^* \ )$$
- Does this really model an operator joystick?

Robot Control System: ITL – 5 (232)

- **Operator control**: The operator is active if at least one of the issued steering commands is not zero.

  OCS $\triangleq$ $(\text{Opactive} = (\text{Loc} \neq 0 \vee \text{Roc} \neq 0) \ )^*$

Robot Control System: Tempura – 1 (233)

```
/*
 * tempura code for robot control system
 * OCS : operator control system
 * ICS : infrared control system
 * MCS : motor control system
 */

define move(X,Y) = {
  format("sending left %t  right %t \n", X, Y)
}.

define s = 0.
define f = 20.
define b = -20.
define mvl = [b,s,b,s,f,f,f,b].
define mvr = [b,b,f,f,f,s,b,s].

/* run */ define robot() = {
  exists  Loc, Roc, Lic, Ric, Opactive, Iractive, Ircounts  :
    { list(Ircounts,8) and stable(struct(Ircounts)) and
```

Robot Control System: Tempura – 2 (234)

```
 define OCS() = {
   while true do {
     len(1) and
     input Loc and input Roc and
     Opactive = {if (Loc ~= 0) or (Roc ~= 0) then 1 else 0}
   }
 } and

define sum(X) = {
  {if Ircounts[0]>100 then X[0] else 0} +
  {if Ircounts[1]>100 then X[1] else 0} +
  {if Ircounts[2]>100 then X[2] else 0} +
  {if Ircounts[3]>100 then X[3] else 0} +
  {if Ircounts[4]>100 then X[4] else 0} +
  {if Ircounts[5]>100 then X[5] else 0} +
  {if Ircounts[6]>100 then X[6] else 0} +
  {if Ircounts[7]>100 then X[7] else 0}
 } and
```

Robot Control System: Tempura – 3 (235)

```
define ICS() = {
  while true do {
    len(1) and input Ircounts and
    Iractive = {if (exists i < 8 : Ircounts[i]>100) then 1 else 0}
    and Lic = sum(mvl) and Ric = sum(mvr)
  }
 } and

define MCS() = {
   while true do {
    len(1) and
    if Opactive=1 and Iractive=0 then {
      format("operator move \n") and move(Loc,Roc) }
    else {
      if Iractive=1 then {
        format("infrared move \n") and move(Lic,Ric) }
      }
  }
 } and
```

Robot Control System: Tempura – 4 (236)

```
 MCS() and ICS() and OCS()
 }
}.
```

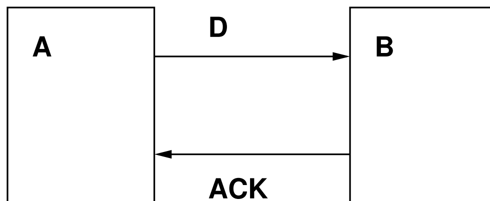
Alternating Bit Protocol: Intro – 1 (237)

- Implementations of the message-passing model, such as TCP/IP, provide the programmer with an abstraction of a stream of data messages sent from process A to process B such that each message is delivered in the order sent.
- However, the physical communication between A and B is not necessarily completely reliable. Due to congestion in the network (other traffic), transient noise, buffer overflows, or other problems, some of the messages sent out over the network from process A may not actually arrive at process B, or they may arrive incomplete or corrupted (usually detected with a checksum).

Alternating Bit Protocol: Intro – 2 (238)

- Therefore, it is necessary to include in these message-passing implementations a protocol that ensures that when a message is lost, it is retransmitted by the sender.
- At the same time, the protocol must guarantee that duplicate messages are not delivered to the receiver and that the delivery order is consistent with the sending order.
- We will study the alternating bit protocol, a simple yet effective protocol for managing the retransmission of lost messages.

Alternating Bit Protocol: Intro – 3 (239)



- ***A***: sender process
- ***B***: receiver process
- ***D***: channel to send data from ***A*** to ***B***
- ACK: channel to send acknowledgements from ***B*** to ***A*** indicating whether data received was corrupted or not.

Protocol managing retransmissions

1. when message is lost then retransmission by the sender
2. sending order is consistent with receiving order

Alternating Bit Protocol: Intro – 4 (240)

For sender A and receiver B the alternating bit protocol is as follows

- Each data message sent by ***A*** contains a protocol bit, 0 or 1.
- When ***A*** sends a message, it sends it repeatedly (with its corresponding bit) until receiving an acknowledgement (ACK) from ***B*** that contains the same protocol bit as the message being sent.
- When ***B*** receives a message, it sends an ACK to ***A*** and includes the protocol bit of the message received. The first time the message is received, the protocol delivers the message for processing. Subsequent messages with the same bit are simply acknowledged.
- When ***A*** receives an acknowledgement containing the same bit as the message it is currently transmitting, it stops transmitting that message, flips the protocol bit, and repeats the protocol for the next message.

Alternating Bit Protocol V1 – 1 (241)

- Introduce the alternating bit protocol in stages
- Both ***D*** and ACK are not lossy

Tempura code:

```
load "tam".
/* tam provides an api for defining channels and
   provides the synchronous receive and send commands:
   chan(C)   : defines a channel C
   wrc(C,X)  : send the value of X over channel C
   rdc(C,V)  : receive the value V over channel C
   ready(C)  : there is data in channel C
*/
```

Alternating Bit Protocol V1 – 2 (242)

```
/* run */ define abp() = {
  exists D, Ack : {
  /* D   : channel used to send and receive data
     Ack: channel used to send and receive acknowledgements
   */

  /* main body of the system, declare 2 channels and
     let the sender and receiver communicate via those channels */
  len(20) and chan(D) and chan(Ack) and
  sender(D,Ack) and receiver(D,Ack)
  }
}.
```

Alternating Bit Protocol V1 – 3 (243)

```
/* the sender */
define sender(D,Ack) = {
  exists Sendbuff, Data, RAck : {
    /* Sendbuff: variable containing data to be sent,
       Data    : variable used in the generation of data
       RAck    : variable used to store the received Ack
    */
    define no_data = -2 and /* indicates no data in Sendbuff */

    /* function indicating whether Sendbuff contains data or not */
    define new_data() = { Sendbuff ~= no_data } and

    /* generate data and put it in Sendbuff */
    define produce() = {
      format("A is producing                          %t\n", Data)
      and  Data := Data + 1 and Sendbuff:= Data
    } and
```

Alternating Bit Protocol V1 – 4 (244)

```
/* send data in Sendbuff via channel D */
define sendD() = {
  format("A is sending    via D   %t\n", Sendbuff) and
  wrc(D,Sendbuff)
} and

/* receive Ack and put it in RAck */
define recAck() = {
  rdc(Ack,RAck) and
  format("A is receiving  via Ack %t\n",RAck) and
  Sendbuff  := no_data
} and

/* initial state: Sendbuff contains no data and Data is zero */
 Sendbuff = no_data and  Data = 0 and

/* show in each state the value of Sendbuff */
always format("A, Sendbuff=%d\n",Sendbuff) and
```

Alternating Bit Protocol V1 – 5 (245)

```
      /* main body of the sender */
      chopstar{ skip and
        if new_data() then { sendD() and recAck() and stable(Data) }
                      else { produce() }
      }
    }
  }.
```

Alternating Bit Protocol V1 – 6 (246)

```
/* the receiver */
define receiver(D,Ack) = {
  exists RData : {
   /* RData: used to store the received data from channel D */

   /* receive data from channel D and store it in RData */
   define recD() = {
     rdc(D,RData) and
     format("B is receiving  via D   %t\n",RData) and
     format("B is consuming                          %t\n", RData)
   } and

   /* send 1 via channel Ack */
   define sendAck() = {
     wrc(Ack, 1) and
     format("B is sending    via Ack %t\n",1)
   } and
```

Alternating Bit Protocol V1 – 7 (247)

```
     /* main body of the receiver */
     chopstar{ skip and
        if ready(D) then { recD() and sendAck() }
     }
    }
  }.
```

Alternating Bit Protocol V2 – 1 (248)

- ***D*** is lossy and ACK is not lossy

Tempura code:

```
load "tam".
/* tam provides an api for defining channels and
   provides the synchronous receive and send commands:
   chan(C)   : defines a channel C
   wrc(C,X)  : send the value of X over channel C
   rdc(C,V)  : receive the value V over channel C
   ready(C)  : there is data in channel C
*/
```

Alternating Bit Protocol V2 – 2 (249)

```
/* run */ define abp() = {
  exists D, Ack : {
   /* D   : lossy channel used to send and receive data
      Ack: channel used to send and receive acknowledgements
   */

  /* main body of the system, declare 2 channels and
     let the sender and receiver communicate via those channels */
  len(20) and chan(D) and chan(Ack) and
  sender(D,Ack) and receiver(D,Ack)
  }
}.
```

Alternating Bit Protocol V2 – 3 (250)

```
/* the sender */
define sender(D,Ack) = {
 exists Sendbuff, Data, RAck : {
    /* Sendbuff: variable containing data to be sent,
       Data    : variable used in the generation of data
       RAck    : variable used to store the received Ack
    */
    define no_data = -2 and /* indicates no data in Sendbuff */

    define faulty_s_data = -1 and /* indicates faulty sent data */

    /* function indicating whether Sendbuff contains data or not */
    define new_data() = { Sendbuff ~= no_data } and

    /* generate data and put it in Sendbuff */
    define produce() = {
      format("A is producing                           %t\n", Data)
      and  Data := Data + 1 and Sendbuff:= Data
    } and
```

Alternating Bit Protocol V2 – 4 (251)

```
/* send data in Sendbuff via lossy channel D */
define sendD() = {
  format("A is sending    via D   %t\n", Sendbuff) and
  if Random mod 2 = 1 then { wrc(D,Sendbuff) }
                      else { wrc(D,faulty_s_data) }
} and

/* receive Ack and put it in RAck
   if RAck is zero then Sendbuff needs to be sent again */
define recAck() = {
  rdc(Ack,RAck) and
  format("A is receiving  via Ack %t\n",RAck) and
  if RAck ~= 0 then {
       Sendbuff := no_data and
       format("A, Ack is ok \n")
  } else {
       Sendbuff := Sendbuff and
       format("A, Ack is not ok, resending \n")
  }
 } and
```

Alternating Bit Protocol V2 – 5 (252)

```
/* initial state: Sendbuff contains no data and Data is zero */
Sendbuff = no_data and  Data = 0 and

/* show in each state the value of Sendbuff */
always format("A, Sendbuff=%d\n",Sendbuff) and

/* main body of the sender */
chopstar{ skip and
  if new_data() then { sendD() and recAck() and stable(Data) }
                 else { produce() }
}
 }
}.
```

Alternating Bit Protocol V2 – 6


```
/* the receiver */
define receiver(D,Ack) = {
 exists RData : {
 /* RData: variable used to store the received data from channel D */
  define faulty_r_data = -1 and /* indicates faulty received data */
  /* receive data from lossy channel D and store it in RData
     if data is not faulty send 1 via channel Ack
     if data is faulty send 0 via channel Ack */
  define recDsendAck() = {
    rdc(D,RData) and format("B is receiving via D %t \n", RData) and
    if RData ~= faulty_r_data then {
      format("B is consuming %t \n", RData)
      and format("B is sending via Ack %t \n",1) and
      wrc(Ack, 1)
    } else {
       format("B, Data over D is lost \n") and
       format("B is sending via Ack %t\n",0) and
       wrc(Ack, 0)
    }
  } and
```

Alternating Bit Protocol V2 – 7


```
    /* main body of the receiver */
    chopstar{ skip and
           if ready(D) then { recDsendAck() }
    }
   }
  }.
```

Alternating Bit Protocol V3 – 1

- Both ***D*** and ACK are lossy

Tempura code:

```
load "tam".
/* tam provides an api for defining channels and
   provides the synchronous receive and send commands:
   chan(C)   : defines a channel C
   wrc(C,X) : send the value of X over channel C
   rdc(C,V) : receive the value V over channel C
   ready(C) : there is data in channel C
*/
```

Alternating Bit Protocol V3 – 2


```
/* run */ define abp() = {
  exists D, Ack : {
   /* D   : lossy channel used to send and receive data
      Ack: lossy channel used to send and receive acknowledgements
   */

  /* main body of the system, declare 2 channels and
     let the sender and receiver communicate via those channels */
  len(20) and chan(D) and chan(Ack) and
  sender(D,Ack) and receiver(D,Ack)
  }
}.

define flip(X) = { if X=1 then 0 else 1 }.
```

Alternating Bit Protocol V3 – 3 (257)

```
/* the sender */
define sender(D,Ack) = {
 exists Sendbuff, Data, RAck, Cs : {
  /* Sendbuff: variable containing data to be sent,
     Data     : variable used in the generation of data
     RAck     : variable used to store the received Ack
     Cs       : variable containing the current protocol bit
   */
  define no_data = -2 and /* indicates no data in Sendbuff */
  define faulty_s_data = [-1,-1] and /* indicates faulty sent data */
  define faulty_r_ack = -1 and /* indicates faulty received ack */

  /* function indicating whether Sendbuff contains data or not */
  define new_data() = { Sendbuff ~= no_data } and
  /* generate data and put it in Sendbuff
     update the protocol bit in Cs */
  define produce() = {
    format("A is producing                       %t\n", Data) and
    Cs := flip(Cs) and Data := Data + 1 and Sendbuff:= Data
  } and
```

Alternating Bit Protocol V3 – 4 (258)

```
/* send data in Sendbuff via lossy channel D */
  define sendD() = {
    format("A is sending     via D   %t\n", [Sendbuff,Cs]) and
    if Random mod 2 = 1 then { wrc(D,[Sendbuff,Cs]) }
                        else { wrc(D,faulty_s_data) }
  } and
```

Alternating Bit Protocol V3 – 5 (259)

```
  /* receive Ack and put it in RAck
     if RAck is not faulty and is equal to protocol bit Cs
     then Sendbuff is updated
     if RAck is not faulty and is not equal to protocol bit Cs
     then Sendbuff needs to be sent again
     if RAck is faulty then Sendbuff needs to be sent again
   */
  define recAck() = {
    rdc(Ack,RAck) and
    format("A is receiving  via Ack %t\n",RAck) and
    if RAck ~= faulty_r_ack then {
      if Cs = RAck then {
        Sendbuff := no_data and format("A, Ack is ok \n")
      } else {
        Sendbuff:=Sendbuff and format("A, Ack is not ok, resending\n")
        }
    } else {
        Sendbuff:=Sendbuff and format("A, Ack is lost, resending \n")
     }
  } and
```

Alternating Bit Protocol V3 – 6 (260)

```
  /* initial state: Sendbuff contains no data and Data is zero
     protocol bit Cs is zero */
  Sendbuff = no_data and  Data = 0 and Cs = 0 and

  /* show in each state the value of Sendbuff and protocol bit Cs*/
  always format("A, Sendbuff=%d, Cs=%d\n",Sendbuff,Cs) and

  /* main body of the sender */
  chopstar{ skip and
    if new_data() then { sendD() and recAck() and
                         stable(Data) and stable(Cs) }
                  else { produce() }
  }
 }
}.
```

Alternating Bit Protocol V3 – 7 (261)

```
/* the receiver */
define receiver(D,Ack) = {
  exists RData, Cr : {
   /* RData: variable used to store the received data
             from channel D
      Cr   : variable containing the current protocol bit
    */

   define faulty_r_data = -1 and /* faulty received data */
   define faulty_s_ack = -1 and /* indicates faulty sent ack */

   /* send over lossy channel Ack the value X */
   define lossy_ack_send(X) = {
     if Random mod 2 = 1 then { wrc(Ack, X) }
                         else { wrc(Ack,faulty_s_ack) }
   } and
```

Alternating Bit Protocol V3 – 8 (262)

```
/* receive data from lossy channel D and store it in RData
   if received data is not faulty and equal to protocol bit Cr
   then send via Ack protocol bit Cr and update Cr
   if received data is not faulty and
      is not equal to protocol bit Cr
   then send via Ack the complement of protocol bit Cr
   if received data is faulty then send via Ack the complement of
   protocol bit Cr
 */
 define recDsendAck() = {
   rdc(D,RData) and
   format("B is receiving  via D   %t\n",RData) and
   /* !!! provide the missing bit here */
 } and
```

Alternating Bit Protocol V3 – 9 (263)

```
   /* initialise protocol bit Cr to 1 */
   Cr = 1 and

   /* show in each state the value of protocol bit Cr */
   always format("B, Cr=%d \n",Cr) and

   /* main body of the receiver */
     chopstar{ skip and
        if ready(D) then { recDsendAck() }
                    else { stable(Cr) }
     }
    }
}.
```

Summary (264)

- A Tempura program is an executable first order ITL formula
- A Tempura program consists of locations, expressions and statements
- The Tempura interpreter constructs a satisfying interval for a Tempura program

Exercises – 1

Exercise 32

Given the following Tempura programs and determine whether they are executable or not. Try to change those that are not executable in such a way that they become executable.

- **define** $\texttt{test_1}() =$ **exists** A : {
  {$A = 0$ **and empty**};
  skip ; {$A = 1$ **and empty**}
  }.
- **define** $\texttt{test_2}() =$
  exists A, B : {$A = 0$ **and** $B = 0$ **and** $A :=$ **next** $(B) + 1$}.
- **define** $\texttt{test_3}() =$ **exists** A :
  {$A = 0$ **and** A **gets** $A + 1$ **and sometimes** $(A = 5)$}.
- **define** $\texttt{test_4}() =$
  exists A : {**empty and** $(A = 0$ **or** $A = 1)$}.

Exercises – 2

Exercise 33

- *Write a Tempura program that ask for the input of a number in the first state and output the square of it in the second state. The program should only have two states.*
- *Write a Tempura program that adds a list of numbers. Show the intermediate "add" results.*
- *Write a Tempura program that consists of two processes. The first process asks for a number in the odd numbered states and output the square of it in the next even numbered state. The second process asks for a number in the even numbered states and output the cube of it in the next odd numbered state.*

Exercises – 3

Exercise 34

Change the Liquid Tank V2 example so that the outlet pipe can be controlled by the operator via an interrupt $\boldsymbol{C}$*, i.e., when* $\boldsymbol{C = 0}$ *then* $\boldsymbol{V2}$ *should be closed. The controller unit should maintain the level of the liquid in the tank within the specified limits while honouring the operator interrupts.*

- *Tempura code version 1*
- *Tempura code version 2*

Exercises – 4

Exercise 35

- *Change the Robot Control System so that priority is given to the operator.*
- *Change the Robot Control System so that the Infra-red motor commands stay within* $\mathbf{-20}$ *and* $\mathbf{20}$ *boundaries.*
- *Tempura code of the Robot Control System*

Exercises – 5

Exercise 36

Complete the Alternating Bit Protocol (ABP) program for lossy Data and Ack channel.

- *TAM library*
- *ABP version 1*
- *ABP version 2*
- *ABP version 3*
- *ABP version 3 output*

Exercises – 6

Exercise 37

Write a Tempura program that models the following supermarket: A supermarket consists of a buying area, *where the customers collect the items they wish to purchase, and one or more* pay tills, *where the customers queue up to pay for those items. Each pay till generally has only one queue of customers. A customer can join any queue and move freely among queues (because all pay tills generally offer the same service). A customer may only join the rear of a queue.*

Exercises – 7

Exercise 37

(continued)
The basic properties of the supermarket and its pay tills and customers are summarised as follows:

Location of clients*:*
A customer may be in any of the following locations:

- *Outside the supermarket.*
- *In the buying area.*
- *In a queue waiting to pay.*
- *At a pay till.*

Exercises – 8

Exercise 37

(continued)
Movements of clients.
A customer may make the following moves:

- *From outside the supermarket to the buying area. This is not possible if the supermarket is closing.*
- *From the buying area to outside. This is possible if either the required item is not available or the queues are too long!*
- *From the buying area to the rear of the queue at some cash point. This is only possible if the pay till is not closing.*

Exercise 37

(continued)

- *From the queue to the buying area (this is only possible if the customer is not currently being served).*
- *From the front of the queue to the pay till associated to the queue. This is only possible if the pay till is free and not closed.*
- *From any position in the queue to the rear of another queue. This is only possible if the new queue is not closing.*
- *From a pay till to outside the supermarket.*

Exercise 37

(continued)

Status of the pay till.

There are three states:

- *Closed (i.e., unable to serve).*
- *Open (i.e., able to serve customers).*
- *Closing (i.e., able to serve customers currently in the queue but not new customers).*

A pay till which is not closed may additionally be in either of the following sub states:

- *Busy (currently serving a customer).*
- *Free (currently not serving a customer).*

Exercise 37

(continued)

Operation of pay tills:

- *Open.*
- *Close queue.*
- *Close.*
- *Begin processing (of a customer).*
- *End processing (of a customer).*

Exercise 37

(continued)

Status of supermarket:

- *Open.*
- *Closed.*
- *Closing.*

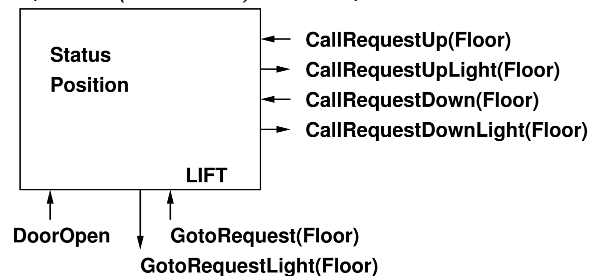
Operation on supermarket:

- *Open.*
- *Close entrance.*
- *Close.*

Exercises – 13

Exercise 38

Write a Tempura program that models one or more lifts, with interactive input of (simulated) button presses.



The Tempura program should have the following parameters:

- `numberOfLifts`, *the number of lifts.*
- `numberOfFloors`, *the number of floors.*

Exercises – 14

Exercise 38

(continued)
The lift shall respond in a timely fashion to the following button presses:

- `CallRequestUp(Floor)`, *from a person waiting on floor* `Floor` *wishing to go up.*
- `CallRequestDown(Floor)`, *from a person waiting on floor* `Floor` *wishing to go down.*
- `GotoRequest(Floor)`, *from a lift passenger.*
- `DoorOpen`, *from a lift passenger (keeps door open if already open, opens door if in the process of closing),*

Exercises – 15

Exercise 38

(continued)
If a lift is moving in a given direction, then it continues to move in that direction until there are no more unserviced requests in that direction. But it shall stop at any floors along the way that have either unserviced `CallRequests` *for the same direction or unserviced* `GotoRequests`*. That is, suppose the lift is moving in the direction of floor k to service a request for that floor. If, on the way to floor k, the lift is about to pass a floor j (not equal to k) such that there is an unserviced call request to floor j, then the lift shall stop at floor j if and only if either*

Exercises – 16

Exercise 38

(continued)

- $k > j$ *and there is an unserviced* `CallRequestUp(Floor)` *with* `Floor`$= j$, *or*
- $k < j$ *and there is an unserviced* `CallRequestDown(Floor)` *with* `Floor`$= j$, *or*
- *there is an unserviced* `gotoRequest(Floor)` *with* `Floor`$= j$.

Exercises – 17

Exercise 38

(continued)
The design should make the following assumptions:
(the unit interval consists of two states, i.e., it is skip*)*

- *Moving from one floor to another floor is done within the unit interval.*
- *The door are kept open for a unit interval.*
- *Opening and closing of the doors is done within the unit interval.*

Exercises – 18

Exercise 38

(continued)
Model the lift as a machine with states and transitions.
The lift can be in the following states:

- `[stop, open]`, *the lift has received no requests and the doors are open.*
- `[up, close]`, *the direction of the lift is up and the doors are closed.*
- `[up, open]`, *the direction of the lift is up and the doors are open.*

Exercises – 19

Exercise 38

(continued)

- `[down, close]`, *the direction of the lift is down and the doors are closed.*
- `[down, open]`, *the direction of the lift is down and the doors are open.*

The lift starts in state `[stop, open]` *and the lift can move from any state to any state.*

Exercises – 20

Exercise 38

(continued)
In order to define the transitions from state `[up, close]` *and* `[up, open]` *the following definitions are needed:*

- `stop_up`*: there is either a* `GotoReq` *or* `CallReqUp` *for the current floor.*
- `called_up`*: there are* `CallReqUp` *for floors that are above the current floor.*
- `called_down`*: there are* `CallReqDown` *for floors that are below the current floor.*

Exercise 38

(continued)
The transitions from state `[up, close]` *are then as follows:*

- *to* `[up, close]` *when* $\neg$***stop_up*** $\wedge$ ***called_up***. *There are no* `GotoReq` *and* `CallReqUp` *requests for the current floor and there are* `CallReqUp` *requests for floors that are above the current floor.*
- *to* `[up, open]` *when* ***stop_up*** $\wedge$ ***called_up***. *There is either a* `GotoReq` *or* `CallReqUp` *request for the current floor and there are* `CallReqUp` *requests for floors that are above the current floor.*

Exercise 38

(continued)

- *to* `[down, open]` *when* ***stop_up*** $\wedge$ $\neg$***called_up*** $\wedge$ ***called_down***. *There is either a* `GotoReq` *or* `CallReqUp` *request for the current floor and there are* `CallReqDown` *requests for floors that are below the current floor.*

Exercise 38

(continued)

- *to* `[stop, open]` *when* ***stop_up*** $\wedge$ $\neg$***called_up*** $\wedge$ $\neg$***called_down***. *There is either a* `GotoReq` *or* `CallReqUp` *request for the current floor and there are no* `CallReqUp` *requests for floors that are above the current floor and there are no* `CallReqDown` *requests for floors that are below the current floor.*
- *to* `[down, close]` *when* $\neg$***stop_up*** $\wedge$ $\neg$***called_up***. *There are no* `GotoReq` *and* `CallReqUp` *requests for the current floor and there are no* `CallReqUp` *requests for floors that are above the current floor.*

Exercise 38

(continued)
The transitions from state `[up, open]` *are as follows*

- *to* `[up, open]` *when* ***called_up*** $\wedge$ ***DoorOpen***.
- *to* `[up, close]` *when* ***called_up*** $\wedge$ $\neg$***DoorOpen***.
- *to* `[down, open]` *when* $\neg$***called_up*** $\wedge$ ***called_down*** $\wedge$ ***DoorOpen***.
- *to* `[down, close]` *when* $\neg$***called_up*** $\wedge$ ***called_down*** $\wedge$ $\neg$***DoorOpen***.
- *to* `[stop, open]` *when* $\neg$***called_up*** $\wedge$ $\neg$***called_down***.

The transitions from the other states should be defined in a similar fashion.

Exercise 39

The Mutual Exclusion Problem for 2 processes is defined as follows:

2 processes are executing in an endless loop a sequence of instructions which can be divided into subsequences: the Critical and the Non-Critical section. The mutual exclusion property is that instructions from the Critical sections of these two processes must not be interleaved.

Exercise 39

(continued)
The solution will be describe by inserting into the endless loop additional instructions that are to be executed by a process wishing to enter and leave its critical section, the Pre and Post sections.

while true do **{**
  ***Non_Critical_Section*;**
  ***Pre_Section*;**
  ***Critical_Section*;**
  ***Post_Section*}**

A process may halt in its Non-Critical section but it may not halt in the Pre, Post and Critical sections. If a process halts in its Non-Critical section, it must not interfere with the other process.

Exercise 39

(continued)
The solution must satisfy the following:

The two processes must not deadlock. If the two process are trying to enter their Critical Section and no process ever succeeds in making the transition from the Pre Section to the Critical Section then we say that the two processes are deadlocked.

There must be no starvation of one of the processes. If a process indicates its intention to enter the Critical Section by entering the Pre Section, eventually it will succeed.

Exercise 39

(continued)
In the absence of contention for the Critical section, a single process wishing to enter its Critical Section will succeed and with minimal overhead.

Exercise 39

(continued)

Examine the following solution by giving its Tempura version and simulate various interesting runs.

Let $\boldsymbol{Turn}$ *be a shared variable with possible values* $\mathbf{1}$ *and* $\mathbf{2}$, *initially it is* $\mathbf{1}$.

```
while true do {                      while true do {
   Non_Critical_Section_1;              Non_Critical_Section_2;
   while Turn ≠ 1 do skip;              while Turn ≠ 2 do skip;
   Critical_Section_1;                  Critical_Section_2;
   Turn := 2}                           Turn := 1}
```


Exercise 39

(continued)

Examine the following solution by giving its Tempura version and simulate various interesting runs.

Let $\boldsymbol{C_i}$ *be shared variable with possible values* $\mathbf{0}$ *and* $\mathbf{1}$, *and that can only be set by process* $\boldsymbol{i}$, *initially* $\boldsymbol{C_i} = \mathbf{1}$.

```
while true do {                      while true do {
   Non_Critical_Section_1;              Non_Critical_Section_2;
   while C2 ≠ 1 do skip;                while C1 ≠ 1 do skip;
   C1 := 0;                             C2 := 0;
   Critical_Section_1;                  Critical_Section_2;
   C1 := 1}                             C2 := 1}
```


Exercise 39

(continued)

Examine the following solution by giving its Tempura version and simulate various interesting runs.

Let $\boldsymbol{C_i}$ *be shared variable with possible values* $\mathbf{0}$ *and* $\mathbf{1}$, *and that can only be set by process* $\boldsymbol{i}$, *initially* $\boldsymbol{C_i} = \mathbf{1}$.

```
while true do {                      while true do {
   Non_Critical_Section_1;              Non_Critical_Section_2;
   C1 := 0;                             C2 := 0;
   while C2 ≠ 1 do skip;                while C1 ≠ 1 do skip;
   Critical_Section_1;                  Critical_Section_2;
   C1 := 1}                             C2 := 1}
```


Exercise 39

(continued)

Examine the following solution by giving its Tempura version and simulate various interesting runs.

Let $\boldsymbol{C_i}$ *be shared variable with possible values* $\mathbf{0}$ *and* $\mathbf{1}$, *and that can only be set by process* $\boldsymbol{i}$, *initially* $\boldsymbol{C_i} = \mathbf{0}$. *Let* $\boldsymbol{Last}$ *be a shared variable with possible values* $\mathbf{1}$ *and* $\mathbf{2}$, *initially* $\boldsymbol{Last} = \mathbf{1}$.

Exercise 39

(continued)

```
while true do {                      while true do {
  Non_Critical_Section_1;              Non_Critical_Section_2;
  C1 := 1;                             C2 := 1;
  Last := 1;                           Last := 2;
  while C2 ≠ 0 ∧ Last = 1              while C1 ≠ 0 ∧ Last = 2
    do skip;                             do skip;
  Critical_Section_1;                  Critical_Section_2;
  C1 := 0}                             C2 := 0}
```


Exercise 40

In distributed computing, it is often useful to be able to designate one and only one process as the coordinator of some activity. This selection of a coordinator is known as the leader election problem.

In this exercise you will consider the problem of electing a leader among a group of 10 processes whose communication pattern is arranged in a ring, where each process can only receive messages from its left neighbour and may send messages only to its right neighbour. It is assumed that each process starts out with a unique identifier (an integer).

Implement and test the following two leader election algorithms. Read over the entire assignment before starting.

Exercise 40

(continued)

Le Lann-Chang-Roberts Leader Election: This is perhaps the simplest of all distributed algorithms. It sends around 10^2 messages in the worst case. Try to construct an execution in which you would get this worst case behaviour.

The idea of this algorithm is that each process starts by passing its own identifier to its neighbour. Whenever a process receives an identifier from its neighbour, it either:

Exercise 40

(continued)

- *passes it on around the ring, if the id received is greater than its own,*
- *discards it, if the id received is less than its own, or*
- *declares itself the leader, if the id received matches its own.*

Notice that the process with the greatest identifier is elected. Create a Tempura program that implements this algorithm.

Exercise 40

(continued)

Peterson Leader Election Algorithm: This algorithm for leader election in a ring is a bit more involved, but its worst case message complexity is much better. See if you can figure out the message complexity of this algorithm. (Hint: notice that at least half of the processes become relays after each round of communication.)

Exercise 40

(continued)

The algorithm works as follows:

1. *In the first phase, each process sends its identifier two steps clockwise (so each process sees the identifiers of its neighbour and its next-to-last neighbour).*
2. *Each process **P** makes a decision based on its own identifier and the identifiers it has received. If **P**'s immediate neighbour's identifier is the greatest of the three, then **P** remains "active" in the algorithm and adopts its neighbour's identifier as its own. Otherwise, P becomes a "relay".*

Exercise 40

(continued)

3. *Now, each active process sends its identifier to its next two active neighbours, clockwise around the ring. Relay processes simply pass along each message they receive.*
4. *Steps 2 and 3 continue repeatedly, with more and more processes dropping out as "relays" until a process finds that its own immediate neighbour is itself, in which case everyone else has dropped out and it declares itself the leader.*

Exercise 40

(continued)

Implement this algorithm as a Tempura Program.
Think about how you will manage sending a message two steps around the ring of active processes. Also, print out useful messages in order to demonstrate that your algorithm is working properly. For example, at each round, active processes should print out the round number, the messages received from the two neighbours, and the decision of whether to remain active or become a relay.

Exercises – 41

Exercise 41

Write a Tempura program that simulates a cash machine (atm). First write an informal specification and then implement it as a Tempura program.

Part VII

AnaTempura

Overview

Introduction

Runtime verification

Application

Runtime Verification Research

Summary

Exercises

Introduction – 1

Assuring the correctness of hardware and software systems, like digital circuits and communications protocols, is a difficult task.

- Complexity of the system.
- Size of the system.
- Requirements that need to be satisfied.

Introduction – 2 (309)

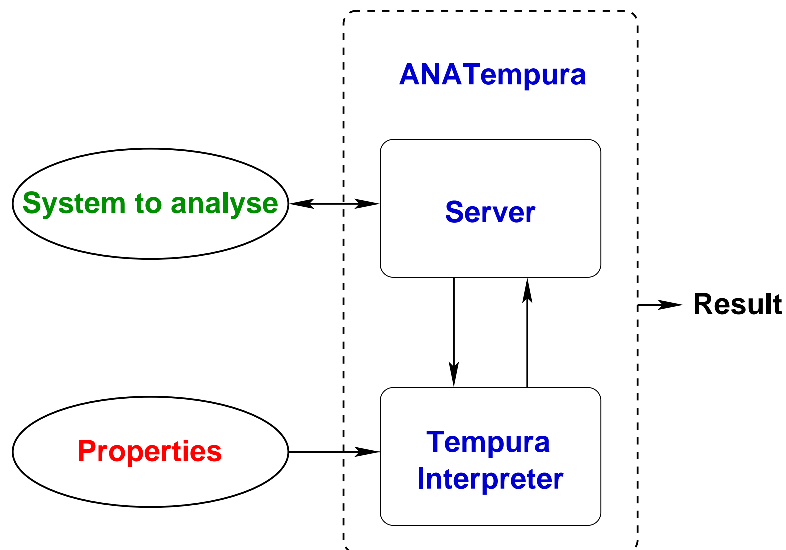
Several techniques have been proposed to assure the correctness of systems:

- testing (simulators, debuggers, etc.),
- formal methods, (theorem provers, proof checkers and model checkers),
- runtime verification (simulators + theorem checkers).

Introduction – 3 (310)

- System: collection of communicating agents (processes)
- State: variables and communication links
- Behaviour: sequence of states (interval)
- Property: set of behaviours

Runtime Verification – 1 (311)



Runtime Verification – 2 (312)

1. Establish all desirable system properties (functional, timing, resource, . . .)
2. Insert at suitable places in the source code of the system assertion points
3. Use AnaTempura to check that the generated behaviour satisfies the desired properties

Runtime Verification – 3

Assertion points → behaviour.

Example: Computation of factorial.

```
main()
{ long y, fac=1;
  assertion("fac", fac);
   text_out("Enter the seed:");
   scanf("%d",&y);assertion("y", y);
   while (y>1) {
     fac=fac*y;
     assertion("fac", fac);
     y=y-1;
     assertion("y", y);
   }
}
```

Runtime Verification – 4

Will generate the following sequence of changes (seed is 4):

```
!PROG: assert fac:1:!
!PROG: assert y:4:!
!PROG: assert fac:4:!
!PROG: assert y:3:!
!PROG: assert fac:12:!
!PROG: assert y:2:!
!PROG: assert fac:24:!
!PROG: assert y:1:!
```

The corresponding behaviour (interval):

y:	?	4	4	3	3	2	2	1
fac:	1	1	4	4	12	12	24	24

Runtime Verification – 5

- Location of assertion points is determined by the variables used in our property of interest.
- Simple search through the source code will locate all places where the variables are changing.
- We will place the assertion points directly after those "places of change".
- This will ensure that behaviour generated during the runtime of the system is indeed the "correct" behaviour.

Runtime Verification – 6

The property we want to check:
(Invariant of the while loop)

```
 define fac_rel(Y,seed) = {
    if Y=seed then Y
    else Y * fac_rel(Y+1,seed)   }.
```

The check:

```
define test() = {
  exists Fac,Y,seed : {
    { seed = 1+random mod 9 and prog_send(seed) };
    check(1,Fac,Y);
    if seed>1 then {
     while Y>1 do check(fac_rel(Y,seed),Fac,Y)
    } else { empty  }
    }
}.
```

Runtime Verification – 7 (317)

Where

```
define check(Z,X,Y) = {
    { skip and get_var("fac",X) and stable(X) and
      if X=Z then {
          format("!:: Pass Prog %7d Prop %7d \n", X, Z)
      } else {
          format("!:: Fac: Prog %d Prop %d \n", X, Z)
      }
    };
    { skip and get_var("y",Y) and stable(X) and stable(Y)};
    { skip and stable(Y) }
}.
```

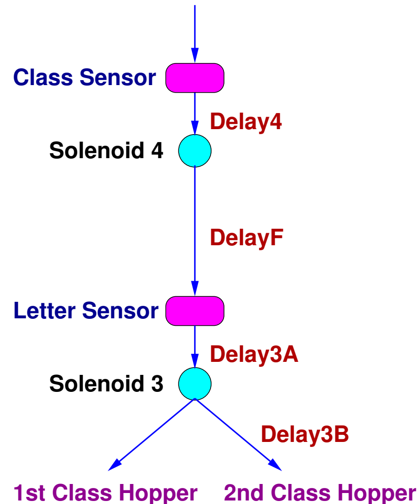
Runtime Verification – 8 (318)

Sample test run (seed is 10):

```
Tempura 1> run test().
!:: Pass Prog       1 Prop       1
!:: Pass Prog      10 Prop      10
!:: Pass Prog      90 Prop      90
!:: Pass Prog     720 Prop     720
!:: Pass Prog    5040 Prop    5040
!:: Pass Prog   30240 Prop   30240
!:: Pass Prog  151200 Prop  151200
!:: Pass Prog  604800 Prop  604800
!:: Pass Prog 1814400 Prop 1814400
!:: Pass Prog 3628800 Prop 3628800
Done!  Computation length:  30.
Total Passes:  31.
Total reductions: 4164(4164 successful).
Maximum reduction depth:  31.
```

Letter Sorter – 1 (319)

Letter sorter: 2 sensors and 2 solenoids



Delay4	70ms
DelayF	250ms
Delay3A	70ms
Delay3B	80ms

Letter Sorter – 2 (320)

```
scan_csensor (&class_sensor);
assertion("class", class_sensor);
if (class_sensor < 2)
   { SolOff(4); Delay(delay4,1);
     SolOn(4);  Delay(delayF,4);
     scan_lsensor (&letter_sensor);
     assertion("lsens",letter_sensor);
     if ( !YellowSet )
       { Delay(delay3A,2); SolOff(3);
         Delay(delay3B,3); YellowSet = 1; }
   } else {
     SolOff(4); Delay(delay4,1);
     SolOn(4); Delay(delayF,4);
     scan_lsensor (&letter_sensor);
     assertion("lsens",letter_sensor);
     if ( YellowSet )
       { Delay(delay3A,2); SolOn(3);
         Delay(delay3B,3); YellowSet = 0; }  }
```

Letter Sorter – 3 (321)

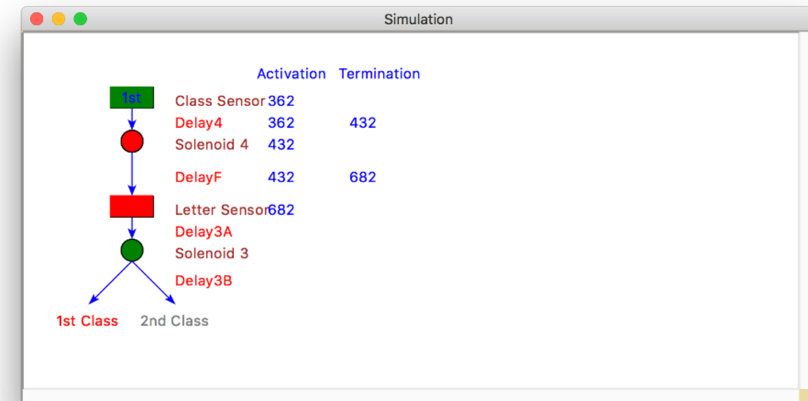
Time is modelled as a variable:

```
!  0:: Solenoid 4 ON: Pass
! 10:: Class sensor 0: Pass
! 10:: Letter sensor 2: Pass
! 20:: Class sensor 1: Pass
! 20:: Solenoid 4 OFF: Pass
. . .
```

corresponds to following interval:

Time:	0	10	10	10	20	20	20
class sensor:			0	0	0	1	1
letter sensor:				2	2	2	2
solenoid 4:	1	1	1	1	1	1	0

Letter Sorter – 4 (322)



Runtime Verification Research – 1 (323)

- David Smallwood. *ITL Monitor: Compositional Runtime Analysis with Interval Temporal Logic*, PhD thesis, in progress.
- Nayef Alhmoud. *Specification and Run-time Verification of a L2 Cache Controller of Multi-core Processor using Interval Temporal Logic*, PhD thesis, in progress.
- Bader Alouffi, *Run-time Verification of Hybrid Systems*, PhD thesis, 2016.
- Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. *Runtime-Monitoring for Industrial Control Systems*. In: Electronics 4.4 2015.
- Sulaiman Al Amro. *Behaviour-based Virus Analysis and Detection*, PhD thesis, 2013.
- Emad Shafie. *Runtime Detection and Prevention for Structure Query Language Injection Attacks*, PhD thesis, 2013.

Runtime Verification Research – 2 (324)

- Turki Mohammed Alghamdi. *Policy-based Runtime Tracking for E-learning Environments*, PhD thesis, 2012.
- Amin Mohammed El-kustaban. *Studying and Analysing Transactional Memory Using Interval Temporal Logic and AnaTempura*, PhD thesis, 2012.
- Helge T. Janicke. *The Development of Secure Multi-Agent Systems*, PhD thesis, 2007.
- François Siewe. *A Compositional Framework for the Development of Secure Access Control Systems*, PhD thesis, 2005.
- Monika Solanki. *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Services*, PhD thesis, 2005.
- Shikun Zhou. *Compositional Framework for the Guided Evolution of Time-Critical Systems*, PhD thesis, 2003.

Runtime Verification Research – 3 (325)

- Jordan Dimitrov. *Formal Compositional Design of Mixed Hardware/Software Systems with semantics of Verilog HDL*, PhD thesis, 2002.

The ITL homepage contains additional links to ITL related research.

Summary (326)

- Runtime verification is an integration of testing and formal verification.
- Our runtime verification approach doesn't suffer from the state explosion problem but it doesn't perform complete tests.

Exercises (327)

Exercise 42

Find out what the following binary does:

- *Windows binary*
- *Linux 32 bit binary*
- *Linux 64 bit binary*
- *Macosx binary*

by feeding it certain input. Change the Tempura code s o that it will pass all tests, i.e., change the following two lines:

```
{Z:=Z+1 and check_z(Z) and stable(I)};
{I:=I+1 and check_i(I) and stable(Z)}
```