Semantic Formalisms: an overview

Eric Madelaine
eric.madelaine@sophia.inria.fr

INRIA Sophia-Antipolis
Oasis team

Mastère Réseaux et Systèmes Distribués
TC4
Program of the course:  
1: Semantic Formalisms

• Semantics and formal methods:  
  – motivations, definitions, examples

• Denotational semantics: give a precise meaning to programs  
  – abstract interpretation

• Operational semantics, behaviour models: represent the complete behaviour of the system  
  – CCS, Labelled Transition Systems
Goals of (semi) Formal Methods

• Develop programs and systems as mathematical objects
• Represent them (syntax)
• Interpret/Execute them (semantics)
• Analyze / reason about their behaviours (algorithmic, complexity, verification)
• In addition to debug, using exhaustive tests and property checking.
Software engineering (ideal view)

• **Requirements** informal
  – User needs, general functionalities.
  – incomplete, unsound, open

• **Detailed specification** formal?
  – Norms, standards?..., at least a reference
  – Separation of architecture and function. *No ambiguities*

• **development**
  – Practical implementation of components
  – Integration, deployment

• **Tests (units then global)** vs verification?
  – Experimental simulations, certification
User requirements → Specification

Programming reuse?

Product → Test & Validation

Cycle of refinements

Component integration unit testing

Increasing cost

V cycle (utopia)
Benefits from formal methods?
automatisation?
Support UML (aparté)

- Notation standardisée, une profusion de modèles/diagrammes :
  - class diagrams
  - use-case diagrams
  - séquence diagrams
  - statecharts et activity charts
  - deployment diagrams

- + stéréotypes pour particulariser les modèles (UML-RT, Embedded UML, …)

- Sémantique ? Flot de conception et méthodologie?
Developer Needs

• Notations, syntax
  – textual
  – graphical (charts, diagrams…)

• Meaning, semantics
  – Non ambiguous signification, executability
  – interoperability, standards

• Instrumentation analysis methods
  – prototyping, light-weight simulation
  – verification
How practical is this?

- Currently an utopia for large software projects, but:
  - Embedded systems
    - Safety is essential (no possible correction)
  - Critical systems
    - Safety, human lives (travel, nuclear)
    - Safety, economy (e-commerce, cost of bugs)
    - Ligne Meteor, Airbus, route intelligente
    - Safety, large volume (microprocessors)
      - Panne réseau téléphonique US, Ariane 5
      - Safety, large volume (microprocessors)
        - Bug Pentium
Industry succes-stories

• Model-checking for circuit development
  – Finite systems, mixing combinatory logics with register states
• Specification of telecom standards
• Proofs of Security properties for Java code and crypto-protocols.
• Certification of embedded software (trains, aircrafts)
• Synthesis ?
Semantics: definition, motivations

• Give a (formal) meaning to words, objects, sentences, programs...

Why?
• Natural language specifications are not sufficient
• A need for understanding languages: eliminate ambiguities, get a better confidence.
• Precise, compact and complete definition.
• Facilitate learning and implementation of languages
Formal semantics, Proofs, and Tools

• Manual proofs are error-prone!
• Tools for Execution and Reasoning
  – semantic definitions are input for meta-tools
• Integrated in the development cycle
  – consistent and safe specifications
  – requires validation (proofs, tests, …)
• Challenge:
  Expressive power versus executability...
Concrete syntax, Abstract syntax, and Semantics

• **Concrete syntax:**
  – scanners, parsers, BNF, ... many tools and standards.

• **Abstract syntax:**
  – operators, types, $\Rightarrow$ *tree representations*

• **Semantics:**
  – based on abstract syntax
  – static semantics: typing, analysis, transformations
  – dynamic: evaluation, behaviours, ...

  *This is not only a concern for theoreticians: it is the very basis for compilers, programming environments, testing tools, etc...*
Static semantics : examples

Checks non-syntactic constraints

- compiler front-end :
  - declaration and utilisation of variables,
  - typing, scoping, … static typing => no execution errors ???

- or back-ends :
  - optimisers

- defines legal programs :
  - Java byte-code verifier
  - JavaCard: legal access to shared variables through firewall
Dynamic semantics

• Gives a meaning to the program (a semantic value)
• Describes the behaviour of a (legal) program
• Defines a language interpreter

|- e -> e’

let i=3 in 2*i -> semantic value = 6

• Describes the properties of legal programs
The different semantic families (1)

- **Denotational semantics**
  - mathematical model, high level, abstract

- **Axiomatic semantics**
  - provides the language with a theory for proving properties / assertions of programs

- **Operational semantics**
  - computation of the successive states of an abstract machine.
Semantic families (2)

• **Denotational semantics**
  – defines a model, an abstraction, an interpretation
    ⇒ *for the language designers*

• **Axiomatic semantics**
  – builds a logical theory
    ⇒ *for the programmers*

• **Operational semantics**
  – builds an interpreter, or a finite representation
    ⇒ *for the language implementors*
Semantic families (3) relations between:

• denotational / operational
  – implementation correct wrt model

• axiomatic / denotational
  – completeness of the theory wrt the model
Program of the course:
1: Semantic Formalisms

• Semantics and formal methods:
  – motivations, definitions, examples

• Denotational semantics: give a precise meaning to programs
  – abstract interpretation

• Operational semantics, behaviour models: represent the complete behaviour of the system
  – CCS, Labelled Transition Systems
Denotational semantics

• Gives a mathematical model (interpretation) for any program of a language.
  All possible computations in all possible environments
  Examples of domains:
  lambda-calculus, high-level functions, pi-calculus, etc...

• Different levels of precision: hierarchy of semantics, related by abstraction.

• When coarse enough
  => effectively computable (finite representation)
  (automatic) static analysis.
Abstract Interpretation

• **Motivations** :
  – Analyse complex systems by reasoning on simpler models.
  – Design models that preserve the desired properties
  – Complete analysis is undecidable

• **Abstract domains** :
  – abstract properties (sets), abstract operations
  – Galois connections: relate domains by adequate abstraction/concretisation functions.
Abstract Interpretation (2)

• Example:
  – Program with 2 integer variables X and Y
  – Trace semantics = all possible computation traces (sequences of states with values of X and Y)
  – Collecting semantics =
    (infinite) set of values of pairs <x,y>
  – Further Abstractions:
    Signs: \( \mathbb{N} \rightarrow \{-,0,+\} \)
    
    succ --> - --> \{-,0\}
    0 --> +
    + --> +
Abstract Interpretation (3)

(a) [In]finite Set of Points

\{ \ldots, \langle 5, 7 \rangle, \ldots, \\
\langle 13, 21 \rangle, \ldots \}\}

(b) Sign Abstraction

\{ x \geq 0 \}
\{ y \geq 0 \}

(c) Interval Abstraction

\{ x \in [3, 27] \\
y \in [4, 32] \}

(d) Simple Congruence Abstraction

\{ x = 5 \text{ mod } 8 \\
y = 7 \text{ mod } 9 \}
Abstract Interpretation (4)

- Function Abstraction: \( F# = \gamma \circ F \circ \alpha \)
Abstract Interpretation (5)

- **Galois connections:**
  - a pair of functions \((\alpha, \gamma)\) such that:

\[
\begin{align*}
\mathbf{L}^\#, \subseteq^\# & \xrightarrow{\gamma} \mathbf{L}^b, \subseteq^b \\
\mathbf{L}^b, \subseteq^b & \xleftarrow{\alpha} \mathbf{L}^\#, \subseteq^\#
\end{align*}
\]

(abstract) \hspace{1cm} (concrete)

- where:
  - \(\subseteq^\#\) and \(\subseteq^b\) are information orders
  - \(\alpha\) and \(\gamma\) are monotonous
  - \(\alpha (\nu^b) \subseteq^\# \nu^\# \iff \nu^b \subseteq^b \gamma (\nu^\#)\)
Abstract Interpretation (6)

example

Java / ProActive code

Data abstraction

Abstract ProActive code

Compilation

Method Call Graph

Operational semantics

Network of Parameterized LTSs

Finite instanciation

Network of finite LTSs

Consistent Chain of approximations
Abstract Interpretation

Summary:
- From Infinite to Finite / Decidable

- library of abstractions for mathematical objects
- information loss: chose the right level!
- composition of abstractions
- sound abstractions:
  property true on the abstract model => true on concrete model
- but incomplete:
  abstract property false => concrete property may be true

Ref: Abstract interpretation-based formal methods and future challenges, P. Cousot, in “informatics 10 years back, 10 years ahead”, LNCS 2000.
Program of the course:  
1: Semantic Formalisms

- Semantics and formal methods:  
  - motivations, definitions, examples
- Denotational semantics: give a precise meaning to programs  
  - abstract interpretation
- Operational semantics, behaviour models: represent the complete behaviour of the system  
  - CCS, Labelled Transition Systems
Operational Semantics (Plotkin 1981)

• Describes the computation
• States and configuration of an abstract machine:
  – Stack, memory state, registers, heap...
• Abstract machine transformation steps
• Transitions: current state -> next state

Several different operational semantics
Natural Semantics : big steps (Kahn 1986)

- Defines the results of evaluation.
- Direct relation from programs to results
  \[ \text{env} \vdash \text{prog} \rightarrow \text{result} \]
  - env: binds variables to values
  - result: value given by the execution of prog

Reduction Semantics : small steps

describes each elementary step of the evaluation
- rewriting relation: reduction of program terms
- stepwise reduction: \(<\text{prog}, s> \rightarrow <\text{prog'}, s'>\)
  - infinitely, or until reaching a normal form.
Differences: small / big steps

• Big steps:
  – abnormal execution: add an « error » result
  – non-terminating execution: problem
    • deadlock (no rule applies, evaluation failure)
    • looping program (infinite derivation)

• Small steps:
  – explicit encoding of non termination, divergence
  – confluence, transitive closure - > *
Natural semantics: examples (big steps)

• Type checking:
Terms: X | tt | ff | not t | n | t1 + t2 | if b then t1 else t2
Types: Bool, Int

• Judgements:
  Typing: $\Gamma |- P : \tau$
  Reduction: $\Gamma |- P \Rightarrow v$
Deduction rules

Values and expressions:

\[ \Gamma |- tt : \text{Bool} \]
\[ \Gamma |- ff : \text{Bool} \]
\[ \Gamma |- \text{true} \]
\[ \Gamma |- \text{false} \]
\[ \Gamma |- t1 + t2 : \text{Int} \]
\[ \Gamma |- t1 : \text{Int} \]
\[ \Gamma |- t2 : \text{Int} \]
\[ \Gamma |- n1 + n2 \]
Deduction rules

• Environment:

\[ \delta :: \{x \rightarrow v\} |- x \Rightarrow v \quad \delta :: \{x : \tau\} |- x : \tau \]

• Conditional:

\[
\Gamma |- b \Rightarrow true \quad \Gamma |- e1 \Rightarrow v
\]

\[
\Gamma |- \text{if } b \text{ then } e1 \text{ else } e2 \Rightarrow v
\]

Exercice : typing rule ?
Operational semantics: big steps for reactive systems

Behaviours

• Distributed, synchronous/asynchronous programs: transitions represent communication events
• Non terminating systems
• Application domains:
  – telecommunication protocols
  – reactive systems
  – internet (client/server, distributed agents, grid, e-commerce)
  – mobile / pervasive computing
Synchronous and asynchronous languages

- Systems build from communicating components: parallelism, communication, concurrency

- Asynchronous Processes
  - Synchronous communications \((\text{rendez-vous})\)
    Process calculi: CCS, CSP, Lotos
  - Asynchronous communications \((\text{message queues})\)
    SDL modelisation of channels

- Synchronous Processes \((\text{instantaneous diffusion})\)
  Esterel, Sync/State-Charts, Lustre

Exercice: how do you classify ProActive?
CCS

• Parallel processes communicating by Rendez-vous:

\[ a? : b! : \text{nil} \xrightarrow{a?} b! : \text{nil} \xrightarrow{b!} \text{nil} \]

\[ a? : P \parallel a! : Q \xrightarrow{\tau} P \parallel Q \]

• Recursive definitions:

\[
\text{let rec } \{ \text{st0} = a? : \text{st1} + b? : \text{st0} \} \text{ in st0}
\]
CCS : behavioural semantics (1)

nil (or skip)

\[ a : P \xrightarrow{a} P \]

\[ P \xrightarrow{a} P' \quad P + Q \xrightarrow{a} P' \]

\[ Q \xrightarrow{a} Q' \quad P + Q \xrightarrow{a} Q' \]
CCS : behavioural semantics (2)

Emissions & réceptions
are dual actions

τ invisible action
(internal communication)

Emissions & réceptions
are dual actions

τ invisible action
(internal communication)

\[ [\mu X. P/X]P \xrightarrow{a} P' \]

\[ \mu X. P \xrightarrow{a} P' \]

\[ P \xrightarrow{a} P' \]

\[ P \xrightarrow{a} P' \]

\[ Q \xrightarrow{a} Q' \]

\[ Q \xrightarrow{a?} Q' \]

\[ P \parallel Q \xrightarrow{a} P' \parallel Q \]

\[ Q \parallel Q \xrightarrow{a} P \parallel Q' \]

\[ P \parallel Q \xrightarrow{\tau} P' \parallel Q' \]

\[ P \parallel Q \xrightarrow{a} P' \parallel Q \]

\[ \mu X.P \xrightarrow{a} P' \]

\[ P \xrightarrow{a} P' \]

\[ P' \notin \{b?, b!\} \]

\[ local \ b \ in \ P \xrightarrow{a} local \ b \ in \ P' \]
Derivations
(construction of each transition step)

\[
\begin{align*}
(a?::P || Q) || a!:R & \xrightarrow{\tau} (P || Q) || R \\
\end{align*}
\]
Example: Alternated Bit Protocol

Hypotheses: channels can loose messages

Requirement: the protocol ensures no loss of messages

Write in CCS?
Example: Alternated Bit Protocol (2)

- **emitter** =

  let rec \{em0 = \text{ack1}?:em0 + \text{imss}?:em1
  
    and em1 = \text{in0}!:em1 + \text{ack0}?:em2
  
    and em2 = \text{ack0}?:em2 + \text{imss}?:em3
  
    and em3 = \text{in1}!:em3 + \text{ack1}?:em0
  
  \}

  in em0

- **ABP** = local \{\text{in0}, \text{in1}, \text{out0}, \text{out1}, \text{ack0}, \text{ack1}, \ldots\}

  in emitter || \text{Fwd\_channel} || \text{Bwd\_channel} || \text{receiver}
Example: Alternated Bit Protocol (3)

Channels that lose and duplicate messages (in0 and in1) but preserve their order?

• Exercise:
  1) Draw an automaton describing the loosy channel behaviour
  2) Write the same description in CCS
Bisimulation

• **Behavioural Equivalence**
  
  - non distinguishable states by observation:
    
    two states are equivalent if for all possible action, there exist equivalent resulting states.

• **minimal automata**

  quotients = canonical normal forms
Some definitions

• **Labelled Transition System (LTS)**
  \[(S, s0, L, T)\]
  where: 
  \(S\) is a set of states
  \(s0 \in S\) is the initial state
  \(L\) is a set of labels
  \(T \subseteq S \times L \times S\) is the transition relation

• **Bisimulations**
  \(R \subseteq S \times S\) is a bisimulation iff
  – It is an equivalence relation
  – \(\forall (p, q) \in R,\)
    \(\exists q'/ (q, l, q') \in T\) and \((p', q') \in R\)
  \(\sim\) is the coarsest bisimulation
  2 LTS are bisimilar iff their initial states are in \(\sim\)
Bisimulation (3)

• More precise than trace equivalence:

• Congruence for CCS operators:

\[ \text{for any CCS context } C[.], \quad C[P] \sim C[Q] \iff P \sim Q \]

Basis for compositional proof methods
Bisimulation (4)

- Congruence laws:
  
  \[ P_1 \sim P_2 \Rightarrow a:P_1 \sim a:P_2 \quad (\forall P_1, P_2, a) \]
  
  \[ P_1 \sim P_2, \ Q_1 \sim Q_2 \Rightarrow P_1 + Q_1 \sim P_2 + Q_2 \]
  
  \[ P_1 \sim P_2, \ Q_1 \sim Q_2 \Rightarrow P_1 \parallel Q_1 \sim P_2 \parallel Q_2 \]

  Etc...
Bisimulation : Exercice
Next courses

2) Application to distributed applications
   – ProActive : behaviour models
   – Tools : build an analysis platform

3) Distributed Components
   – Fractive : main concepts
   – Black-box reasoning
   – Deployment, management, transformations

www-sop.inria.fr/oasis/Eric.Madelaine

Teaching