Slide 1

An introduction to model checking

University of Alberta
Edmonton
July 3rd, 2002

Guy Tremblay

Dépt. d'informatique
UQAM

Slide 2

Outline

- What are formal specification and verification methods?
- What is model checking?
  - How can the behavior of a reactive system be specified?
  - How can *temporal* properties be specified?
- How can model checking be done?
- Why and how can model checking be done *in parallel*?

1 What are formal methods?

"[Formal methods are] mathematically based techniques used to describe the properties of computing systems. They [are used to] specify, develop, and verify systems in a systematic and rigorous manner [. . .]" [Wing90]

Key elements of a formal method:

Slide 3

- *Formal* language for writing specifications
- Rules to check the quality of the specifications
- Strategies and rules to refine and *verify* the specifications

Foundation on which *everything* rests = Formal specifications

What is a formal *specification* language?

Formal language ⇒ well-defined syntax and semantics:

- Syntax = EBNF, syntax diagrams, etc.
- Semantics = algebras, automatas and transition systems, relations and predicates, etc.

Slide 4

Specification lang. ⇒ describes the external *behavior* of a software component . . .

- by describing its key properties
- in an *abstract* way (without unneeded implementation details)
- without saying *how* it is going to be implemented (*non-algorithmic*)

Thus, a programming language *is not* a specification language because . . .

- it is algorithmic
- it is not abstract (arrays, pointers, etc.)

A specification provides a *non-algorithmic* description
$\Rightarrow$ Describes the "what?" instead of the "how?"

Slide 5

An example (in `Spec`):

```
FUNCTION square_root{ precision: real SUCH THAT precision > 0.0 }

  MESSAGE root( x: real SUCH THAT x >= 0.0 )
    REPLY( r: real )
        WHERE r >= 0.0  &  almost_equal( r^2, x )

  CONCEPT almost_equal( r1 r2: real ) VALUE( b: boolean )
      WHERE b <=> abs(r1 - r2) <= precision
END
```

What are the major benefits of formal specifications and formal methods?

> *"Having to better understand the* specificand *by compeling the analyst to be abstract yet precise about the properties of the system can be more rewarding than having the specification itself."* [Wing90]

- Specifications are more explicit, precise, with less ambiguity.
- *Formalization effort* $\Rightarrow$ help identify errors, ambiguities, and problems *early*.

Slide 6

- Provide a better foundation for implementation work.
- Allows for use of tools (manipulation, analysis, simulation).
- Basis for developing tests.

- Provide a basis for doing <u>formal verification</u>.

Slide 7

Why are there many specif. lang. and methods?

Many different *styles* of specifications:

- Abstract modeling for machines and objects (VDM, Z, Spec, etc.);
- Algebraic specification for ADT (Larch, ACT ONE, etc.);
- Behavioral specification for reactive systems (CCS, CSP, LOTOS, ACP, etc.)
- Safety and liveness properties (modal and temporal logics)
- ...

Similar to programming languages:

- Diverse application domains
- Various styles and paradigms
- Varying expressive power and analyzability

Slide 8

2 Specifying reactive and concurrent systems

A system is said to be *reactive* ...

- when it maintains a *constant* interaction with its environment
- when its behavior is "*event-driven*"

A system is said to be *concurrent*

- when its behavior is determined by the *interaction* of multiple tasks (processes) that cooperate and exchange information

Slide 9

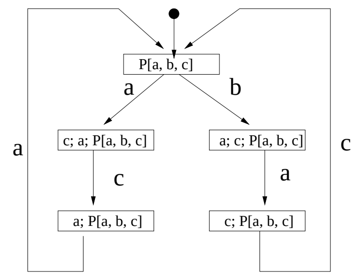
Modeling the behavior of reactive systems

- The behavior of a reactive system can be described by specifying the actions that it can (and cannot) perform
- A computation of a reactive system is generally *infinite*

  $\Rightarrow$ use of labeled transition systems (automata)

A small example: Lotos specification and its graphical description

```
process P[a, b, c]: noexit :=
  a; c; a; P[a, b, c]
  []
  b; a; c; P[a, b, c]
endproc
```



Slide 10

Modeling concurrent systems

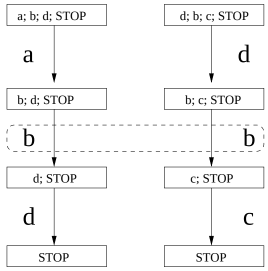
Concurrent behavior can be expressed by *interleaving* semantics:

- Concurrent (unordered) actions can occur in any order

  $\Rightarrow$ any possible interleaving is allowed
- Synchronized actions = actions performed *synchronously* by two (or more) agents

  $\Rightarrow$ only one action visible

```
a; b; d; STOP
|[b]|
d; b; c; STOP
```



Possible set of *visible* runs = $\{$adbdc, adbcd, dabdc, dabcd$\}$

Slide 11

Specifying *properties* of the behavior

- Automata = form of *operational description*
  ≈ describes how to generate the possible sequences of actions
- But . . . such a description does not make explicit the *properties* satisfied by the behavior
  - Safety properties: *nothing bad will ever happen*.
  - Liveness: *something good will eventually happen*.

Different approaches to the specification of properties:

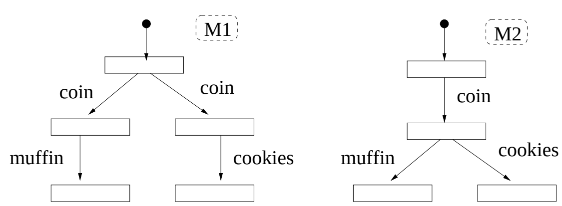
- Modal logic: local properties of current state
- Temporal logic: properties of runs
  - Linear-time logic
  - Branching-time logic

Slide 12

Linear-time logic

Linear-time property = property along a *single* path of execution
⇒ A state satisfies a linear-time logic property if all complete paths that start from this state satisfy the property

Example:



These two machines have the same set of (complete) paths:

`{ coin;muffin, coin;cookies }`

⇒ they will satisfy the same *linear*-time properties

. . . but do they really have the same behavior?

Slide 13

Modal logic

Modal logic = expresses (local) properties of the current state

- Possibility (may): $\langle \mathtt{a} \rangle \phi$

  = it is possible to do action a and then reach a state that satisfies ϕ
- Necessity (must): $[\mathtt{a}]\phi$

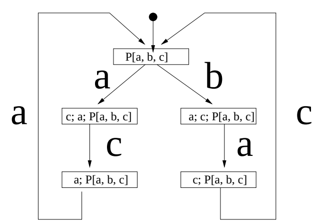
  = whenever action a is done, the resulting state satisfies ϕ

Two typical idioms:

- $\langle \mathtt{a} \rangle \mathtt{tt}$ = it is possible to do a
- $[\mathtt{a}]\mathtt{ff}$ = a cannot be done

Slide 14

Examples on P =



- $\mathtt{P} \models \langle \mathtt{a} \rangle \mathtt{tt}$

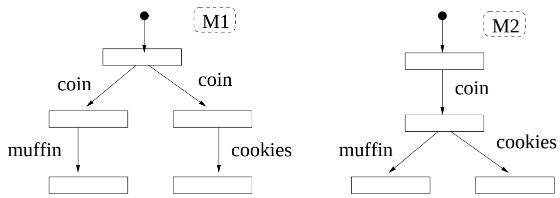
  (Liveness) P can do a as its first move
- $\mathtt{P} \models [\mathtt{a}][\mathtt{b}]\mathtt{ff}$

  (Safety) In its starting state, P cannot do an a followed by a b
- $\mathtt{P} \models [-\mathtt{b}]\langle \mathtt{c} \rangle \mathtt{tt} \wedge [-\mathtt{a}]\langle \mathtt{a}, \mathtt{d} \rangle \mathtt{tt}$

  (Liveness) If the 1st action is not a b, then the 2nd is a c and if the 1st is not an a, the 2nd is an a or a d

Slide 15

Modal logic $\Rightarrow$ Machines M1 and M2 can now be distinguished:



- $\mathtt{M1} \not\models [\mathtt{coin}]\langle\mathtt{muffin}\rangle\mathtt{tt}$
- $\mathtt{M2} \models [\mathtt{coin}]\langle\mathtt{muffin}\rangle\mathtt{tt}$

Slide 16

Temporal (branching-time) logic

Temporal logic

= Expresses properties of the runs (the paths)

$\Rightarrow$ Describes *qualitatively* the occurence of events in time

CTL = Computation Tree Logic:

- $s \models \mathtt{AG}\phi$

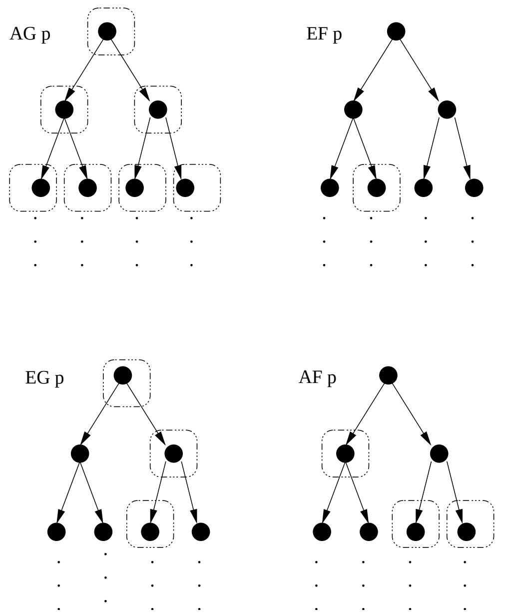
  = ϕ holds on all possible states reachable from state s

  = *Always*(ϕ)
- $s \models \mathtt{EF}\phi$:

  = from s, there exists a path where ϕ eventually holds

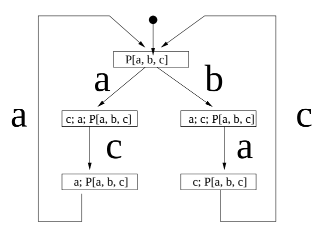
  = *Eventually*(ϕ)

Slide 17



Slide 18

Examples on P =



- $\texttt{P} \models \texttt{AG}([\texttt{b}][\texttt{c}]\texttt{ff})$

  (Safety) For any run, it is never possible to do b followed by c

- $s \models \texttt{AG}(\texttt{EF}\,\langle\texttt{a}\rangle\texttt{tt})$

  (Weak liveness) Along every path starting from s, eventually, an action a will be possible.

Mu-calculus

Modal mu-calculus = A temporal logic with explicit fixpoint operators

Syntax:

$$\phi ::= \texttt{tt} \mid \texttt{ff} \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [L]\phi \mid \langle L \rangle \phi \mid \mu X.\phi \mid \nu X.\phi$$

Slide 19

Always and *Eventually* using fixpoint operators:

$$\begin{aligned} \textit{Always}(\phi) &= \nu X.\phi \wedge [-]X \\ \textit{Eventually}(\phi) &= \mu X.\phi \vee \langle - \rangle X \end{aligned}$$

3 Model checking

Model checking = "*A technique that relies on building a finite model of a system and checking that a desired property holds in that model.*" [ClarkeEtAl96]

Model checking = An *automatic* technique for *verifying* properties of finite state systems

Slide 20

General approach:

1. Construct M = a model (of the behavior of the system)
2. Specify ϕ = a property expected of the system (expressed in modal/temporal logic)
3. Check that M satisfies ϕ. If not, produce counter-examples.

Slide 21

Implementation requires exploration of the state space

$\Rightarrow$ Important requirement for M = must be *finite*

Advantages/disadvantages of model checking (+/-):

+ Verification is completely automatic
+ Can produce *counter-examples* that represent subtle errors

- State explosion problem

Slide 22

Primary applications (so far) = hardware and protocol verification:

- IEEE Futurebus+ cache coherence protocol [McMillan93]
  (a number of previously undetected errors were found)
- ISDN/ISUP telecommunication protocol [Holzmann92]
  (122 errors found)
- HDLC channel controller [DePalmaGla96]
  (uncovered major bug)
- Active structural control system in civil engineering [ElseaidyEtAl96]
  (uncovered major bug that could have worsen effect of vibration)
- ...

4 Implementation of model checking

4.1 Global vs. local model checking

- Global model checking: Given a finite model, M, and a formula, ϕ, *determine the set of states in M that satisfy ϕ.*
- Local model checking: Given a finite model, M, a formula, ϕ, and a state s in M, *determine whether s satisfies ϕ.*

Slide 23

Characteristics of global vs. local model checking:

- Solution to global problem $\Rightarrow$ solution to local one
- Solution to global problem $\Rightarrow$ exploration of the whole state space
- Solution to local $\Rightarrow$ demand-driven exploration of state space

4.2 How to compute fixpoints

Solving model checking problem $\Rightarrow$ need to find solutions to recursive equations.

Let $\langle - \rangle$ and $[-]$ denote the uses of the modalities with arbitrary actions.

Recall that:

- $\mathtt{AG}\phi$ = *Always*(ϕ)
- $\mathtt{EF}\phi$ = *Eventually*(ϕ)

Slide 24

Always and *Eventually* can be defined recursively:

$$\begin{aligned} \textit{Always}(\phi) &= \phi \wedge [-]\textit{Always}(\phi) \\ \textit{Eventually}(\phi) &= \phi \vee \langle - \rangle \textit{Eventually}(\phi) \end{aligned}$$

Slide 25

Definition: x is a fixpoint of f iff $f(x) = x$

Fact: A solution to a recursive equation is *always* a fixpoint of *an appropriate function*.

Example: $x = 2 * x$

- Associated function: $\tau(x) = 2 * x$
- Solution: 0 is a solution since $\tau(0) = 0$

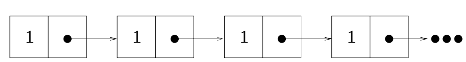
Example: $x = x$

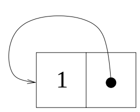
- Associated function: $\tau(x) = x$
- Solution: Any n is a solution since $\tau(n) = n$

Slide 26

Example: a recursive definition of a list of integers

- Equation: $l = 1 : l$
- Associated function: $\tau(l) = 1 : l$
- Solution: Let $ones = [1, 1, 1, 1, \ldots]$ be an infinite list of 1s.
  Then $\tau(ones) = ones$.





Slide 27

Fact: The least solution of a functional τ can be obtained as the limit of a sequence of approximations (where $\perp$ is the least element of the domain):

$$\bigsqcup_{n=0}^{\infty} \tau^n(\perp)$$

Example:

- Let $\tau(l) = 1 :\ l$
- Let $\tau^0(l) = \perp$
- Let $\tau^{i+1}(l) = \tau(\tau^i(l)) = 1\ :\ \tau^i(l)$

$$\begin{array}{rcl}
\tau^0(\perp) & = & \perp \\
\tau^1(\perp) & = & 1 :\ \perp \\
\tau^2(\perp) & = & 1 :\ 1 :\ \perp \\
\ldots & & \\
\tau^{i+1}(\perp) & = & 1 :\ 1 :\ \ldots :\ \perp
\end{array}$$

Slide 28

4.3 Global model-checking for mu-calculus

= Determine set of states satisfying property ϕ

$\approx$ Compute denotational semantics (set of states)

$$\begin{array}{rcl}
[\![\mathtt{tt}]\!]_{\mathcal{V}} & = & \mathcal{P} \\
[\![\mathtt{ff}]\!]_{\mathcal{V}} & = & \{\} \\
[\![X]\!]_{\mathcal{V}} & = & \mathcal{V}(X) \\
[\![\phi_1 \wedge \phi_2]\!]_{\mathcal{V}} & = & [\![\phi_1]\!]_{\mathcal{V}} \cap [\![\phi_2]\!]_{\mathcal{V}} \\
[\![\phi_1 \vee \phi_2]\!]_{\mathcal{V}} & = & [\![\phi_1]\!]_{\mathcal{V}} \cup [\![\phi_2]\!]_{\mathcal{V}} \\
[\![[L]\phi]\!]_{\mathcal{V}} & = & \{p \mid \forall\, a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \Rightarrow p' \in [\![\phi]\!]_{\mathcal{V}}\} \\
[\![\langle L\rangle\phi]\!]_{\mathcal{V}} & = & \{p \mid \exists\, a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \wedge p' \in [\![\phi]\!]_{\mathcal{V}}\} \\
[\![\mu X.\phi]\!]_{\mathcal{V}} & = & fix_{\mu}\ \tau_{\phi,\mathcal{V}} \\
 & & \text{where } \tau_{\phi,\mathcal{V}}(x) = [\![\phi]\!]_{\mathcal{V}[x \mapsto X]}
\end{array}$$

$$fix_\mu\, \tau_{\phi,\mathcal{V}} \quad = \quad \bigcup_{n=0}^{\infty} \tau^n_{\phi,\mathcal{V}}(\{\})$$

Where $\tau^0(x) = x$

$\tau^{i+1}(x) = \tau(\tau^i(x))$

Slide 29

Termination property: Since the model (number of states) is *finite*, a fixpoint will be reached after a finite number of iterations

4.4 Local model-checking for mu-calculus

= Determine whether a state satisfies a property ϕ

$\approx$ Compute axiomatic semantics (inference rules)

- = Set of (inductive) rules that specify if a process p satisfies a formula ϕ

Slide 30

$$\begin{array}{lll}
p & \models & \texttt{tt} \\
p & \not\models & \texttt{ff} \\
p & \models & \phi \wedge \psi \text{ iff } p \models \phi \text{ and } p \models \psi \\
p & \models & \phi \vee \psi \text{ iff } p \models \phi \text{ or } p \models \psi \\
p & \models & [L]\phi \text{ iff } \forall a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \Rightarrow p' \models \phi \\
p & \models & \langle L \rangle \phi \text{ iff } \exists a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \wedge p' \models \phi \\
p & \models & \mu X.\phi \text{ iff } \ldots
\end{array}$$

Slide 31

5 Parallel model checking

5.1 The state explosion problem

Modeling of concurrency by interleaving $\Rightarrow$ Total number of states may grow exponentially with the number of concurrently executing components

Example:

- 100 lines Lotos specification with 10 small processes $\Rightarrow$
  - 56 000 states
  - 180 000 transitions

Global model checking and exhaustive exploration of the state space

$\Rightarrow$ keep state space in memory to avoid multiple exploration of same state

$\Rightarrow$ lot of space required to store the graph (LTS)

Slide 32

Possible solutions to state explosion problem

- Symbolic model checking
- Exploit various kinds of information to reduce the number of states/transitions
  (as long as the key properties are preserved)
- ...
- Use a parallel machine with multiple nodes to provide more memory

Slide 33

5.2 Target machine and environment

Target parallel machine = EARTH (CAPSL, Univ. of Delaware, Newark, DE)

- Fine-grain multi-threaded parallelism = multiple levels of parallelism (threads vs. fibers)
- Irregular dynamic parallelism = data flow style scheduling
- Off-the-shelf computer = EARTH-RTS (Pthreads and sockets)
  - Earthquake = 16-processors Beowulf cluster (University of Delaware)
- Programming language = Threaded-C

Slide 34

CADP toolbox (INRIA, Grenoble, France):

- Translator from Lotos to LTS + numerous other tools:
  - Simulation
  - Equivalence checking
  - Model checking for regular alternation-free mu-calculus
  - . . .
- LTS provides an *implicit* representation of the graph (transition function)
- Goal = construct an explicit representation (state graph)

Slide 35

5.3 Distributing the graph

General strategy for distributing the graph

- Traverse the graph by evaluating the transition function
- Use a *dispersion* function `h` to distribute the states on the various processors
- Handle transition `t = (s1, e, s2)` on processor `h(s2)`
- Never send a transition more than once by keeping track of the states that have been `visited`

Slide 36

Pseudo-code:

```
// Initialization phase in process 0
s0 = start_state();
visited = {s0};
FOREACH transition t = (s0, e, s1) going out of s0 DO
  SEND t TO processor h(s1);
END
```

```
 // Processing phase (on all processors)
 WHILE not terminated (?!) DO
   RECEIVE transition t0 = (s0, e0, s1) from arbitrary process;
   IF !(s1 IN visited) THEN
     visited = visited U {s1};
     FOREACH transition t = (s1, e, s2) going out of s1 DO
        SEND t TO process h(s2);
     END
   END
 END
}
```

5.4 Detecting termination

Key problem = Detecting when all transitions have been processed

Currently implemented solution = Distributed detection termination based on the number of messages sent/received

5.5 Next step = perform model checking

Slide 37

- Currently: Only distribution of transitions has been implemented (graduate course project)
- Still need to add processing associated with model checking itself
  - Global model checking $\Rightarrow$ multiple exploration of the graph (fixpoint computation)
  - Local model checking $\Rightarrow$ demand-driven exploration

6 Conclusion

- Model checking is an interesting approach to formal verification because it is *automatic*
- Major difficulty = need to handle large state space

Slide 38

- On-going and future work:
  - Short-term = see how parallel and distributed execution can help with state explosion problem
  - Long-term = apply model checking to π-calculus
    $\Rightarrow$ handle mobile processes (dynamic and non-finite state space ;(