IF simulation and verification tool for UML

- introduction
- the IF notation
- toolbox architecture
- the UML front-end IFx
- conclusions

VERIMAG
The IF toolbox: objectives

Model-based development of real-time systems

Use of high level modeling and programming languages
- Expressivity for faithful and natural modeling
- Cover functional and extra-functional aspects
- Openness

Model-based validation
- Combine static analysis and model-based validation
- Integrate verification, testing, simulation and debugging

Applications:
Protocols, Embedded systems, Asynchronous circuits, Planning and scheduling
The IF toolbox: approach

Modeling and programming languages (SDL, UML, SCADE, Java …)

IF: Intermediate Format, based on a general and powerful semantic model

Static Analysis

Transition systems

Simulation, test, verification1, verification2, verification3
The IF toolbox: challenges

Find an adequate intermediate representation

**Expressiveness**: direct mapping of concepts and primitives of high modeling and programming languages
- asynchronous, synchronous, timed execution
- buffered interaction, shared memory, method call …

Use information about structure for efficient validation and traceability

**Semantic tuning**: when translating languages to express semantic variation points, such as time semantics, execution and interaction modes
overview

• introduction
• the IF notation
• the IF validation tools
• the UML front-end (IFx)
• conclusions
System description

Processes
extended timed systems
(non-determinism, dynamic creation)

Interactions
asynchronous channels
shared variables

Data
predefined data types
(basic types, arrays, records)
abstract data types

System description

IF simulation and verification tool for UML

Omega Workshop
February 17, 2005
System description

A set of interacting processes

• A process instance:
  – executes asynchronously with other instances
  – can be dynamically created
  – owns local data (public or private)
  – owns a private FIFO buffer

• Inter-process interactions:
  – asynchronous signal exchanges (directly or via signalroutes)
  – shared variables
System description

// processes
process P1(N1)
  ...
endprocess;
...
process P3(N3)
  ...
endprocess;

// signalroutes
signalroute sr1(1) ...
  from P1 to P3 ;

// signals
signal s1(t1)
signal s2(t1, t2),

// local data
Process description

Process = hierarchical timed automaton

```plaintext
process P1(N1);
fpar ...;

// types, variables, constants, procedures
state s0 ...;
    ... // transition t1
endstate;
state s1 #unstable...;
    ... // transitions t2, t3
endstate;

... // states s2, s3, s4
endprocess;
```

local data + local clocks

P1(N1)
The transition equation is given by:

\[ \text{transition} = \text{urgency} + \text{trigger} + \text{body} \]

The state transition diagram shows:

- **State s0**: ...
  - **Urgency**: ...
  - **T1**: ...
    - **Urgency**: eager
      - Provided: \( x \neq 10; \)
      - When: \( c2 \geq 4; \)
      - Input: \( \text{update}(m); \)
      - Body: ...
      - Next State: \( s1; \)
  - **End State**: ...

- **Statement**: data assignment, message emission, process or signal route creation or destruction, ...

The trigger is composed of:

- **Untimed Guard**: ...
- **Timed Guard**: ...
- **Signal Consumption from the Process Buffer**: ...

The trigger expression is:

\[ \text{trigger} = \{ \text{untimed guard} \} \]

The statement list comprises:

- Sequential, conditional, or iterative composition.
Signal routes

signal route = connector = process to process communication channel with attributes, can be dynamically created

signalroute \( s_1(1) \) #unicast #lossy #fifo

from server to client with grant, fail;

attributes:

- queuing policy: \texttt{fifo} | \texttt{multiset}
- reliability: \texttt{reliable} | \texttt{lossy}
- delivery policy: \texttt{peer} | \texttt{unicast} | \texttt{multicast}
- delay policy: \texttt{urgent} | delay[\( l, u \)] | rate[\( l, u \)]
Delivery policies

Peer

server(0)

client(1)

to one specific instance

Unicast

server(0)

client(0)  client(1)  client(2)

to a randomly chosen instance

Multicast

server(0)

client(0)  client(1)  client(2)

to all instances
The model of time [timed automata]
- global time $\rightarrow$ same clock speed in all processes
- time progress in stable states only $\rightarrow$ transitions are instantaneous

\[ \text{system configuration} \]

\[
\begin{align*}
\text{time} &= 0 \\
q_0 &\rightarrow q_1 \rightarrow q_2 \\
\text{time} &= \delta_0 \\
q_3 &\rightarrow q_4 \rightarrow q_5 \\
\text{time} &= \delta_0 + \delta_1
\end{align*}
\]
Timed behavior

- operations on clocks
  - set to value
  - deactivate
  - read the value into a variable

- timed guards
  - comparison of a clock to an integer
  - comparison of a difference of two clocks to an integer

```
state send;
output sdt(self,m,b) to {receiver}0;
set t:= 10;
nextstate wait_ack;
endstate;

state wait_ack;
input ack(sender,c);
...
when 10 <t<20 ;
...
endstate;
```
Dynamic priorities

• priority order between process instances p1, p2 (free variables ranging over the active process set)

\[ \text{priority_rule_name} : p1 < p2 \text{ if condition}(p1,p2) \]

• semantics: only maximal enabled processes can execute

• scheduling policies
  – fixed priority: \( p1 < p2 \) if \( p1 \) instanceof T and \( p2 \) instanceof R
  – run-to-completion: \( p1 < p2 \) if \( p2 = \text{manager}(0).\text{running} \)
  – EDF: \( p1 < p2 \) if \( \text{Task}(p2).\text{timer} < \text{Task}(p1).\text{timer} \)
overview

- introduction
- the IF notation
- the IF validation tools
- the UML front-end (IFx)
- conclusions
IF simulation and verification tool for UML

IF Description

IF Exploration Platform

 Exploration Platform

- TGV test generation
- model construction
- model checking
- guided simulation
- mincost path extraction

Test Suites

- SPIDER
- CADP

LTS

Guided simulation

Schedules
Core components

IF specifications
- parser
- writer

IF AST
- syntactic transformation tools:
  - static analyser
  - code generator

Compilation
- compiler

C/C++ code
- application specific process code
- predefined modules (time, channels, etc.)

Interaction model
- dynamic scheduling

State space representation

LTS exploration tools
- debugging
- model checking
- test generation
Exploration platform

IF simulation and verification tool for UML

Omega Workshop
February 17, 2005
Dealing with Time

Dedicated module
- including clock variables
- handling dynamic clock allocation (set, reset)
- checking timing constraints (timed guards)
- computing time progress conditions w.r.t. actual deadlines and
- fires timed transitions, if enabled

Two implementations for discrete and continuous time
(others can be easily added)

i) discrete time
- clock valuations represented as varying size integer vectors
  - time progress is explicit and computed w.r.t. the next enabled deadline

ii) continuous time
- clock valuations represented using varying size difference bound matrices (DBMs)
  - time progress represented symbolically
  - non-convex time zones may arise because of deadlines: they are represented implicitly as unions of DBMs
Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization

Validation tools

Model-Based Validation
- static analysis
- model checking
- test generation
- optimization
Static analysis

• approach
  – source code transformations for model reduction
  – code optimization methods

• techniques implemented so far
  – live variable analysis: remove dead variables and/or reset variables when useless in a control state
  – dead-code elimination: remove unreachable code w.r.t. assumptions about the environment
  – variable abstraction: extract the relevant part after removing some variables

• usually, impressive state space reduction
Model-checking using observers

- **Observers** are used to specify safety properties in an operational way
- They are described as the processes – specific command for monitoring events, system state, elapsed time
- 3 types of states: normal / error / success
- **Semantics:** Transitions triggered by monitored events and executed with highest priority

```
match output SDT(void, b)
[b <> R(0).flag]
```

```
match input ACK(void)
[x <= t_ack]
```

```
set x := 0
[x >= t_ack]
```

```
match input ACK(void)
[x <= t_ack]
```

```
set x := 0
[x >= t_ack]
```
Behavioral equivalence checking

- **LTS comparison:**
  - equivalence relations (“behavior equality”):
    \[ \text{System} \approx \text{Requirements} \]
  - preorder relations (“behavior inclusion”):
    \[ \text{System} \leq \text{Requirements} \]

- **LTS minimization:**
  - quotient w.r.t an equivalence relation:
    \[ (\text{System} / \approx) \]

- **CADP can be used to check the following relations:**
  - weak/strong bisimulation, branching, safety, trace equivalence
• User defined costs associated to transitions of IF descriptions e.g., execution times

• problem: find the min-cost execution path leading from the initial state to some goal state

• three algorithms implemented:
  – Dijkstra algorithm (best first)
  – A* algorithm (best first + estimation)
  – branch and bound (depth-first)

• applications:
  – job-shop scheduling (find the makespan)
  – asynchronous circuit analysis (find the maximal stabilization time)
overview

• introduction
• the IF notation
• the IF validation tools
• the UML front-end (IFx)
• conclusions
the UML front-end (IFx)

Rose, Rhapsody, Argo, ...

XMI
UML model + time annotations

UML front-end IF

UML-IF frontend GUI
UML2IF translator + compliance checker
UML validation driver

IF model
IF tools

IF simulation and verification tool for UML
Coverage of the Omega-UML profile

- fully OO models: classes with operations, attributes, associations, generalization, statecharts; basic data types; the Omega Action Language (compatible to UML1.4 A.S.)
- UML observers for specifying requirements
- timing constraints

Tool connection

- XMI 1.0 or 1.1 for UML 1.4: Rational Rose, I-Logix Rhapsody and others.
compilation of UML elements

- structure
  - UML object → IF process
  - attributes & associations → variables
  - inheritance: replication of structural features

- behavior
  - state machines, actions → syntactic translation (almost)
  - operations X::m(x, y, …)
    ⇒ one IF process for every invocation of X::m
    
    process X::m(x, y, …)
    - lives the period of activation, implements behavior
    - encapsulates the "stack frame" variables

    ⇒ predefined signals
    
    call_{X::m}, return_{X::m}

- flexible: adaptable to different call semantics (async, with futures…)

IF simulation and verification tool for UML
compilation of method calls

caller

\text{call}_{X::m}(...)

\text{create}

\text{return}_{X::m}(...)

\text{X}

\text{X::m}(...)

\text{X::m}(...)

\text{X::m}(...)

\text{X::m}(...)

polymorphism, concurrency...

polymorphism \(\Rightarrow\) dynamic binding resolved with signals
- the object state machine decides the method implementation which is executed in response to \(\text{call}_X::m\)

concurrency \(\Rightarrow\) activity group management
- each active object has an associated group manager
- it handles/dispatches external calls for objects of the group
- keeps track of the running object

run-to-completion
- implemented with dynamic priority rules
e.g. : \(\forall x,y. (x\text{.manager} = y) \Rightarrow x < y\)
formalizing requirements: UML observers

- special objects monitoring the system state / events
- example (Ariane-5): If the Pyro1 object enters state “Ignition_done”, then the Pyro2 object shall enter the state “Ignition_done” after the time TimeConstants.MN_5*2 + Tpstot_prep and before the time TimeConstants.MN_5*2 + Tpstar_prep.
the Omega timing framework

time is global, external to the system

• imperative constructs: describe time dependent behavior
  – time-related primitive types Time, Duration, 0-ary operator now : Time
  – mechanisms for measuring durations: timers, clocks

• declarative constructs: timed events and constraints
  – orthogonal to the functional specification
    ⇒ separation of modeling concerns
    ⇒ flexible semantics: independent from semantics of the functional part*

  – timed events: history of occurrence times of identified state changes
    • Sending, receiving, consuming a signal, Calling, receiving, accepting, answering an operation
    • Executing an action / a state machine transition, Entering/exiting a state, Creating an object,…

  – constraints on duration between event occurrences
    • Assumptions (taken as hypotheses)
    • Requirements (to be verified)

⇒ TA

⇒ observers
UML-to-IF compiler

Options

- ignore parts of the model
- eager / lazy semantics
- .................
the UML front-end (IFx)

Rose, Rhapsody, Argo, ...

XMI
UML model + time annotations

UML front-end
IF

UML2IF translator + compliance checker

UML validation driver

UML-IF frontend GUI

IF model

IF tools
the front-end GUI

- Wrapper of IF tools $\rightarrow$ (partly) hidden from the UML user
- Functionality :
  - interactive simulation / diagnostics analysis
  - scenario rewind / reply / load /saves
  - source tracing
  - conditional breakpoints
  - UML and other customized views (XSLT stylesheets)
  - batch tool launcher (compilers, verification)
IF simulation and verification tool for UML

Omega Workshop
February 17, 2005
case studies in Omega

Ariane-5 case study (EADS) – developed in Rational Rose
• statically validate the well formedness of the model wrt the Omega profile,
• proved 9 safety properties of the flight regulation and configuration components,
• analyzed the schedulability of the cyclic / acyclic components under the assumption of fixed priority preemptive scheduling policy,
• proved a safety property concerning bus read/write coherence under this policy

MARS case study (NLR) – developed in I-Logix Rhapsody
• static validation
• proved 4 safety properties concerning the correctness of the MessageReceiver,
• discover reactivity limits of the MessageReceiver and to fine-tune its behavior in order to improve reactivity.

Sensor Voting an Monitoring case study (IAI)– developed in Rational Rose
• static validation
• proved 4 safety properties concerning the timing of data acquiring by the three Sensors: end-to-end duration, duration between consecutive reads, etc.
Conclusions

• IF is an unique platform:
  – relates functional and non-functional aspects:
    • distribution, communication, external C/C++ code, dynamic creation,
    • real-time, deadlines, resources, dynamic priorities
  – integrates state of the art validation techniques and tools
  – provides front-end to SDL/UML and related tools

• one step further, component-based modeling and validation
  – combination of synchronous and asynchronous systems
    (e.g, Lustre/Esterel/StateCharts and SDL/UML)
  – composability of models and compositional reasoning
Combination of static analysis and validation techniques proves to be crucial for coping with complexity and broadens the scope of application of the tool e.g.,
- use static analysis for data intensive applications
- use partial order reduction techniques for control intensive applications

The use of high level languages incurs additional costs wrt low level modeling languages
- There is a price to pay for enhanced expressivity and faithful modeling
- Abstraction and simplification can be carried out automatically by static analysis

Observers are a powerful formalisms for safety requirements
- Easy to use by practitioners
- Limitation to safety properties is not a serious one, especially for RT systems