Overview

1. Overview on results developed in the OMEGA project (2002-2005)
   1. Motivation
   1. UML semantic based profile for the expression of real-time properties
   1. Validation tools for real-time properties

2. The IF validation environment (since 1998)
   1. The IF language
   1. Mapping from UML to IF
   1. The tool-set for simulation and verification
   1. Case studies (from Omega)
The IST-2001-33522 Omega project on Correct Development of Real-Time Embedded Systems

- **Duration**: January 2002 – February 2005
- **Aim**: Definition of a development methodology in UML for embedded and real-time systems integrating formal validation techniques
- **Coordinator**: Verimag

---

Omega: Partners and supporters

**Academic (tool and technology providers)**
- Verimag, France – coordinator
- Christian-Albrechts University Kiel, Germany
- CWI (Centrum voor Wiskunde en Informatica), Netherlands
- University of Nijmegen, Netherlands
- OFFIS, Oldenburg, Germany
- Weizmann Institute, Israel

**Users (requirements and case studies)**
- EADS Launch Vehicles, France
- France Telecom R&D, France
- Israeli Aircraft Industries, Israel
- NLR (Nationaal Lucht- en Ruimtevaartlaboratorium), Netherlands

**Supporters (UML tool providers)**
- I-Logix
- Rational Software, IBM
- Telelogic
Validation in the context of model based development of real-time systems

UML model

Requirements (use cases, observers) + TIME

Structure: classes, components, ...

Behavior: state machines, operations, ...

+ TIME

Architecture

Platform - resources ...

Test generation

Simulation

Prototype validate update

PSM

Implementation

Validation tools

UML in the context of model based development

Strong points of UML

1. Support of requirement level and design level notations, including architecture and components, which made their proofs
2. User acceptance
3. Meta-model facilitating syntactic transformations

Weak points of UML (for validation of dynamics)

1. Includes also informal notations
2. Concepts are defined at syntax level, no well defined (dynamic) semantics and no framework for defining one
3. No notion of consistency between different notations
4. Weak support of real-time concepts (improved by SPT profile and UML 2.0)
Overview

1. Overview on the results developed in the OMEGA project
   1. Motivation
   1. UML profile for the expression of real-time properties and its semantics
   1. Validation tools for real-time properties

2. The IF validation environment
   1. The IF language
   1. Mapping from UML to IF
   1. The tool-set for simulation and verification
   1. Case studies (from Omega)

Choice of a UML profile

n Fact: validation is only one aspect, we need a profile for modelling real-time embedded systems
   Do not restrict the considered UML profile too much, so as to make it just match the concepts in the validation tools

n Fact: validation requires semantics
   It is not enough to define keywords (tag values and stereotypes as in SPT or QoS profile) for defining a profile

n Fact: building validation tools is expensive
   Reuse existing state-of-the-art methods and tools
   Be open to many UML tools
   standard exchange format (XMI) and standard UML extension mechanisms
   Be open to different methodologies
   Be open to a variation of semantics
Omega real-time profile for real-time systems

A semantics has been formally defined for this subset and implemented in several tools

- Aim: Consistent validation results by different tools (hard to achieve, in fact there are some variations in the choices made in different tools)

Structure

- Class diagrams
  - Distinguishing active objects (mono-threaded processes) and passive objects (local data)
  - Private and public attributes and methods
  - Several associations with defined interpretation (inheritance, aggregation, navigation)
  - Components define visibility constraints on the sub-components (UML 1.4 has no diagrams for components, little used case studies)

Interaction model: semantics

- active/passive objects define activity groups
- interactions: primitive/triggered operations, asynchronous signals
Omega real-time profile

- **Behavior**
  - **Local Behaviour**
    - State machines
    - Action language (concrete syntax, compatible to UML 1.4 A.S.)
      - Object creation / destruction
      - Interactions (point-to-point): primitive/trIGGERed operations, asynchronous signals
  - **Global behavior**
    - OCL for the expression of state invariants and invariants of event histories
    - Live Sequence Charts = extension of sequence diagrams
    - Observers = state machines triggered by semantic level events (constructive description of a set of event sequences = property)
  - Consistency defined through distinction between operational specification (global liveness), Constraints and Requirements (constraints to be verified)

Omega real time profile: Timing

Compatible SPT profile and UML 2.0

- **Basics**
  - A notion of global time, *external* to the system
  - Time primitive types: Time, Duration with operations
    - **Timed Events**: sequence of instants of occurrences of identified state changes in each execution:
      - “SendSignal”, “ReceiveSignal”, “ConsumeSignal”
      - “InvokeMethod”, ....
      - “EnterState”, “ExitState”
      - “StartAction”, “EndAction”
      - ....
  - Operational time access (as in UML 2.0): *time dependent behavior*
    - Mechanisms for measuring durations: timers, clocks
    - And corresponding actions: set, reset,...
Omega real time profile: Timing

- **Time constraints**
  1. **Constraints on durations** between occurrences of events (OCL based)
     - Patterns for constraining durations between occurrences of 2 events
     - SPT like derived patterns associated with syntactic entities
       - response time,
       - duration of actions, deadline constraints,
       - duration in state,
       - delay of channel, ...
  1. **Observers** with time constraints for the expression of (local and global) properties implying several events

- **Scheduling related**
  1. **Resources** accessed in mut. excl. and consuming execution time
  1. **Execution time of actions**
  1. **Dynamic priorities** for expressing scheduling policies

Time profile: requirements as 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 + Tpsot_prep and before the time TimeConstants.MN_5^2 + Tpsar_prep.***
Conclusions on profile

- Modeling reactive systems
  - Rich subset (similar to Rhapsody, Room, SDL, …)
  - Semantics with non determinism with a sufficient granularity for adding timing
  - Useful extensions
    - Multiway communication (within and in between activity groups)
    - Atomic reaction to a set of events (as in Esterel)
    - Direct modeling of synchronous dataflow

- Real-time
  - Direct expression of time constraints as in existing frameworks
  - Introduce a general naming scheme for events and event occurrences for the expression of arbitrary time constraints
  - A framework for defining semantics of SPT like RT profiles

Overview

1. Overview on the results developed in the OMEGA project
   - Motivation
   - UML semantic based profile for the expression of real-time properties
   - Validation tools for real-time properties

2. The IF validation environment
   - The IF language
   - Mapping from UML to IF
   - The tool-set for simulation and verification
   - Case studies (from Omega)
Validation tools for the Omega profile

Some challenges:

- Possibility to implement a semantics with variation points
- Avoid introduction of state explosion or proof overhead due to the translation of UML concepts to input language of the tool
- Which high level concepts to keep at semantic level and which ones to map into simpler ones?

Omega tool-set

UML CASE tools (+ restrictions)

- XMI Omega exchange format
- XMI extractors
- PVS based proofs
- Parameterized systems
- LSC scenario tools (Weizman)
- Requirements analysis
- UVE (Offis)
- Model-Checking of functional properties
- IF
- Model-checking and simulation of timing properties on timed models

Verification

Tool formats
IF language and validation environment

Marius Bozga
Lucian Ghirvu
Laurent Mounier
Joseph Sifakis

Jean-Claude Fernandez
Susanne Graf
Iulian Ober
Yassine Lakhnech

IF intermediate representation

Not a modelling language, but an exchange format for simulation and validation

Processes (dynamic behavior)
- extended timed automata
- (non-determinism, dynamic creation)

Interactions
- asynchronous channels
- global variables

Execution control
- Priority rules
- “resources” for mutex constraints

Data
- predefined data types
  (basic types, arrays, records)
- abstract data types
IF notation: System description

\[
\begin{align*}
\text{const } & N_1 = \ldots ; & \text{// constants} \\
\text{type } & t_1 = \ldots ; & \text{// types} \\
\text{signal } & s_2(t_1, t_2), & \text{// signals} \\
& \text{// signal routes (types)} \\
& \text{signalroute } sr(1) ... & \text{// route attributes from P1 to P3} \\
& \text{// processes (types)} \\
& \text{process } P_1(N_1) & \ldots & \text{// data + behaviour} \\
& \ldots & & \\
& \text{process } P_3(N_3) & \ldots & \\
& \text{endprocess;} & & \\
& \ldots & & \\
& \text{endprocess;} & & \\
\end{align*}
\]

IF notation: Process description

Process = hierarchical, timed systems with actions

\[
\begin{align*}
\text{process } & P_1(N_1); \\
& \text{fpar } \ldots ; & \text{// parameters} \\
& \text{// types, variables, constants, procedures} \\
& \text{state } s_0 \ldots & \text{// transition t1} \\
& \text{endstate;} & \text{// outgoing transitions} \\
& \text{state } s_1 \text{#nostable} \ldots ; & \text{// transitions t2, t3} \\
& \text{endstate;} & \text{// states } s_2, s_3, s_4 \\
& \text{endprocess;} & \text{// local data + local clocks} \\
\end{align*}
\]
IF notation: transition

transition \( = \) urgency + trigger + body (+cost)

\[
\begin{align*}
\text{state } & s_0 \\
& \ldots \\
\text{t1} & \text{urgency} \\
& \text{eager} \\
& \text{provided } x! = 10; \\
& \text{when } c2 \geq 4; \\
& \text{input update}(m); \\
& \text{body } \ldots \text{...} \\
\text{nextstate } & s_1; \\
& \text{...} \\
\text{endstate;}
\end{align*}
\]

\[
\text{statement } = \text{data assignment} \\
\text{signal sending,} \\
\text{process or signalroute creation or destruction, } \ldots
\]

IF notation: dynamic creation

- process creation:

\[ p := \text{fork client(true)} \]

- process destruction:

\[ \text{kill client(2)} \]

\[ \text{kill } p \]

- process termination:

\[ \text{stop} \]

\[ \text{the "self" instance is destroyed, together with its buffer, and local data} \]
IF notation: Data and types

Variables:
• are *statically typed* (but *explicit conversions* allowed)
• can be declared *public* (= shared)

Predefined basic types: integer, boolean, float, pid, clock

Predefined type constructors:
• (integer) interval: `type fileno = range 3..9;`
• enumeration: `type status = enum open, close endenum;`
• array: `type vector = array[12] of pid`
• structure: `type file = record f fileno; s status endrecord;`

Abstract Data Type definition facilities …

IF notation: interactions - signal routes

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

attributes:
• queuing policy: *fifo* | *multiset*
• reliability: *reliable* | *lossy*
• delivery policy: *peer* | *unicast* | *multicast*
• delay policy: *urgent* | *delay[l,u]* | *rate[l,u]*
IF notation: interactions - delivery policies

- **Peer**:
  - Server (0)
  - Client (1)

- **Unicast**:
  - Server (0)
  - Client (0), Client (1), Client (2)

- **Multicast**:
  - Server (0)
  - Client (0), Client (1), Client (2)

  - To one specific instance
  - To a randomly chosen instance
  - To all instances

IF notation: interactions - signal exchange

**Signal emission (non blocking):**

- **To a specific process**: `output req (3, open) to server(2);`
- **Via a signalroute**: `output req(3, open) via s0(1);`
- **Mixed**: `output token via link(1) to client(k+1)%N;`  
  
  - `k = integer(self)`

**Signal consumption (blocking):**

- `input req (f, s);`
IF notation: System description - example

```plaintext
const NS= ..., NC= ...;
type file= ..., status= ..., reason= ...;

signal stop(), req(file, status), fail(reason), grant(), abort(), update(data);

signalroute s0(1) #multicast
  from server to client with abort;
signalroute s1(1) #unicast #lossy
  from server to client with grant, fail;
signalroute s2(1) #unicast
  from client to server with req;

process server(NS) ... endprocess;
process client(NC) ... endprocess;
```

IF notation: timed behavior

```
- operations on clocks
  - set to value
  - desactivate
  - 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;
```
IF notation: timed behavior

The model of time [timed systems]
1. global time → same clock speed in all processes
2. time progress in (stable) states only → transitions are instantaneous

IF notation: 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

- Express execution modes and scheduling policies
  1. fixed priority: p1 < p2 if p1 \text{ instance of T and p2 \text{ instance of R}} (R,T process types)
  2. EDF: p1 < p2 if Task(p2).deadl-timer < Task(p1).deadl-timer
  3. run-to-completion: p1 < p2 if p2.group = p1.group \land p1.running \land p1 ≠ p2
Overview

1. Overview on the results developed in the OMEGA project
   1. Motivation
   2. UML semantic based profile for the expression of real-time properties
   3. Validation tools for real-time properties

2. The IF validation environment
   1. The IF language
   2. Mapping from UML to IF
   3. The tool-set for simulation and verification
   4. Case studies (from Omega)
(1) Mapping UML-to-IF

Structure:
1. class → process type
2. attributes & associations → variables
3. inheritance → replication of features
4. signals, basic data types → direct mapping

Time:
1. Timers and clocks → direct mapping
2. Time constraints and observers → additional clocks and guards, observers

Behavior
1. state machines (with restrictions) → IF hierarchical automata
2. action language → IF actions, automaton encoding
3. operations:
   - operation call/return → signal exchange
   - operation activation → process creation
   - polymorphism → untyped PIDs
   - dynamic binding → destination object automaton determines the executed procedure

Execution modes
1. dynamic priorities to define run-to-completion execution
   \[ p_1 < p_2 \quad \text{if} \quad p_2.\text{active} = p_1 \]
   \[ p_1 < p_2 \quad \text{if} \quad p_2 = p_1.\text{active.running} \land p_1 \neq p_2 \]
(1) UML-IF frontend

UML tools

IF tools

UML model + time annotations

XMI

UML2IF translator + compliance checker

UML validation driver

IF model

IF static analysis

Timeline variables

Slicing

Abstraction

Time constraint propagation

IF behavior tools

Simulator

Test generator

Scheduling analysis

State explorer

Graph level tools (CADP)

Minimization, comparison, composition...

UML to tools

IF to tools

IF behavioral tools

IF model

IF exporters

IF static analysis

Timeline variables

Slicing

Abstraction

Time constraint propagation

IF behavior tools

Simulator

Test generator

Scheduling analysis

State explorer

Graph level tools (CADP)

Minimization, comparison, composition...

IF toolset architecture

IF (3)

Static Analyzer

IF Description

IF Exploration Platform

External tools

TGV Test Generation

Model construction

Model checking

Interactive + guided exploration

Mincost path extraction

SPIDER

Test Suites

LTS

Observer based On-the-fly verification

Minimization

(1) UML-IF frontend

(2) IF Exploration Platform

(3) IF Static Analyzer

(4) Interactive + guided exploration

(5) Minimization
(2) IF exploration platform

IF description

IF AST

C/C++ code

parser

writer

syntactic transformation tools:
- static analyser
- code generator

application specific process code

predefined modules
(time, channels, etc.)

interleaved execution

dynamic priorities (p.o)

state space representation

LTS exploration tools:
- debugging
- model checking
- test generation

Exploration method: random/guided simulation, on-the-fly exploration (w. p.o reduction), state graph generation,...
(2) IF exploration platform: time implementation

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:
discrete and continuous time

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

IF toolset architecture
(3) IF 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
  - slicing: elimination of code that does not influence a particular property
  - variable abstraction: extract the relevant part after removing some variables

- **Properties**
  - Property preserving transformation
  - usually, impressive state space reduction

(4) Model-checking using observers

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

```
output SDT(void, b)
[b <> R(0).flag]  [b = R(0).flag]
set x := 0

match output SDT(void, b)

idle
match input ACK(void) [x <= t-ack]

test
[b = R(0).flag]  set x := 0

error
match input ACK(void) [x <= t-ack]

wait
```
(5) Behavioral equivalence checking

LTS comparison

1. equivalence relations ("behavior equality"):
   System ≈ Requirements
2. preorder relations ("behavior inclusion"):
   System ≤ Requirements

LTS minimization

1. quotient w.r.t an equivalence relation (System|≈)
Aldebaran can be used to check the following relations:
weak/strong bisimulation, branching, safety, trace equivalence

Min cost execution path for a cost function
Overview

1. Overview on the results developed in the OMEGA project
   1. Motivation
   1. UML semantic based profile for the expression of real-time properties
   1. Validation tools for real-time properties

2. The IF validation environment
   1. The IF language
   1. Mapping from UML to IF
   1. The tool-set for simulation and verification
   1. Case studies (from Omega)

Omega case studies

4 Case studies – 3 from aeronautics – 1 from Telecom

- An ATV: Replicated voting algorithm (2 voters, 3 sensors) (IAI)
  1. Correctness of voting algorithm (functional correctness)
  1. Timing correctness (under work, encouraging preliminary results)

- Mars automatic camera: Error controller for two desynchronized data sources (NLR)
  1. Timely recognition of a bus error

- Ariane 5: flight schedule and control (EADS)
  1. Correct ordering and timely occurrence of sub-phases
  1. Interaction between synchronous and asynchronous part
  1. Scheduling issues (under work)

- Components for a “dépannage” service (just started) (FTRD)
  1. Timely occurrence of events
  1. Constraints on event occurrence orders
Ariane 5 flight controller: 40 minutes flight

- many timers (smallest with 70ms rate)
- 31 objects

Ariane 5 flight controller

Control: navigation, guidance, global timing ...
Regulation: firing control of different stages
Configuration: manages separation of the different stages
Ariane 5: validation steps

- **Translation to IF**: only few elementary data variables, but complex timing aspects (transmission of timer values through signals)

- **Static analysis**
  - **Clock reduction**: 1. Description: 143 timers reduced to 41 clocks
    - 2. Description: 55 timers, no more reduction
  - **Dead variable reduction**: 20% of all variables are dead in each state
  - **Slicing**: elimination of passive processes (without outputs)

- **Model generation**
  - **Appropriate environment restrictions**
  - **Partial order reduction** (31 processes)

- **Verification**
  - Evaluation of properties (observers)
  - Out of 16 properties, 3 did not hold in the initial, carefully debugged model
  - Construction and visualization of bisimulation reduced models

---

Ariane 5: model generation results

<table>
<thead>
<tr>
<th>reduction strategy</th>
<th>states</th>
<th>transitions</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 none</td>
<td>too large !</td>
<td>too large !</td>
<td>?</td>
</tr>
<tr>
<td>2 live</td>
<td>2 200 000</td>
<td>18 700 000</td>
<td>2h07 m</td>
</tr>
<tr>
<td>3 p.o.</td>
<td>2 000</td>
<td>2040</td>
<td>3.21 s</td>
</tr>
<tr>
<td>4 live + p.o.</td>
<td>1 600</td>
<td>1642</td>
<td>1.68 s</td>
</tr>
</tbody>
</table>

Model generation after application of clock reduction and slicing of passive processes
**Ariane 5 case study: verification result**

Property:

*whenever a problem is detected before the ignition command to the main Vulcan engine is sent, then the ignition of the main engine and the secondary engine is stopped.*

Graph obtained from the global model by weak bisimulation minimisation

- Allows visual inspection of the graph

---

**MARS Bus Controller**

Informal spec: timely recognition of bus error

**bus error state** = \( k_1 \) consecutive losses of NavMsg or of AltMsg

**ok again** if \( k_2 \) consecutive NavMsg and \( k_2 \) consecutive AltMsg

- DatabusManager
- ControllerMonitor
  - prevOK : Boolean
  - curOK : Boolean
- MessageReceiver
- DatabusController
- NavigationDataSource
- AltitudeDataSource

\( N \) Desynchronized message sources with period \( C \) and jitter \( j \)
MARS Bus Controller: structure

Mars: a property

Property: if there is no data sent from the Altitude Data Source in a time window of 90ms, then the Message Receiver should be in BusError state at the end of this time window.
Mars: Message receiver

Mars: abstract version of message receiver

Version 1: Recognize every period “signal ok” or “signal loss” and have counters

- in state operational: for consecutive losses
- in state BusError: for consecutive receptions

<table>
<thead>
<tr>
<th>Operational</th>
<th>ControllerError</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reception counting (timer = p + 2 * j)</td>
</tr>
<tr>
<td></td>
<td>2 consecutive good receptions from both sources (both counters &gt;= 2)</td>
</tr>
<tr>
<td></td>
<td>3 consecutive losses from 1 source (one counter =3)</td>
</tr>
<tr>
<td></td>
<td>loss counting (timer = P + 2*j) or P after a loss</td>
</tr>
<tr>
<td></td>
<td>reset counters</td>
</tr>
</tbody>
</table>
Verification

System verifiable for 2 desynchronized data sources and fixed period (25) and jitter (10),
3 sources state explosion. Why is this a hard verification problem?

Senders: arbitrary but never changing offset

\[ C + 2J \leq C \]

Desynchronized Sender and Receiver

Senders: arbitrary but never changing offset

\[ C + 2J \leq C \]

Synchronized Sender and Receiver

Synchronizing Sender and receiver makes the verification much less expensive. But this solution does not resist to even small clock drift.
Mars: abstract version of message receiver

**Version 2: Asymmetric**
- In state operational, look for a “long timeout” (3*C + 2*j) – no need for counter
- In state BusError, no change

- Operational
- BusError
- ControllerError

- Upon two consecutive good receptions from both sources
  (both counters >= 2)

- Upon 3 consecutive losses from one source
  (timer = 3 * C + 2 * j)

- Loss counting
  (timer = C + 2 * j)

- Data reception

Verification results

**Version 1:** Found 2 subtle errors (counters reset at a bad moment + timing)

**Version 2:** surprise, property does not hold, the is system less reactive.

- Error found (in depth 3000)
- Hard to find error by testing
- Model-checking allowed to find and correct the error on hand of a counter example in ½ h

**General:**

- Verification of a parameterized version (n data sources, period C and jitter j) under work using PVS
- The considered UML model is small, but it is enviseageable to extract the same small verification problem from a more complete model
Conclusions: mapping UML-IF

The mapping from UML to IF is very concrete, why not just generate code and execute it?

- There is a notion of *simulation time* and this is crucial
- The flexibility of interactive or guided simulation is much higher than with a tool executing code
  - complete control on the resolution of non determinism
  - possibility to avoid or detect loops
- For exhaustive exploration, execution of code is not enough anyway
- Abstraction can be applied at IF level (dependent on the property) – this is a motivation for the concepts at IF level

Overview on the tool set capabilities

- The translation from UML to IF introduces syntactic overhead, but only linear increase of model size
- Manipulation of model-checker by users with some initial assistance from the tool builders
- Abstraction of data part: possible today in a restricted way separation of data and control part
- Property depending extraction of subsystems: slicing
- Proof of feasibility of integration of formal verification into the development process
Pointers to additional information:

http://www-omega.imag.fr

http://www-verimag.imag.fr/~async/IF