UML-based modeling and verification of real-time embedded systems

Iulian Ober
IRIT, Univ. of Toulouse

OMEGA UML and the MARS case study - joint work with: Marius Bozga, Susanne Graf, Yassine Lakhnech, Joseph Sifakis

Ileana Ober

Yuri Yushtein
Overview

I. About model driven engineering & embedded systems
II. UML : a family of languages
III. OMEGA UML
IV. Timed simulation and model checking
V. A real-life case study
Embedded systems computation under physical constraints

Physical world

Control engineering

Physical constraints

HW engineering

Platform constraints

Requirements

About model driven engineering & embedded systems
UML: a family of languages
OMEGA UML
Timed simulation and model checking with IF
A case study
About models a
Model driven engineering
master the complexity

Exploratory models
- Abstract
- Understandable
- Acurate
- Predictive
- Inexpensive

Productive models
- Abstract
- Understandable
- Acurate
- Predictive
- Inexpensive

the model = the system
System model $\Omega$

Requirements:
- functional
- timing, schedulability, QoS

Logical (or functional) architecture
- logical (de)composition
- assembly
- communication
- structure
- components
- classes
- ... 
- component behavior
- state machines
- operations
- timing

Physical (non-functional) architecture
- tasks
- resources
- scheduling policy
- transaction protection
- ...

Semantic models

Simulation

Validation

Implementation
A bit of history…
modeling in the ’80 – ’90s

• Lots of (slightly different) languages and design techniques
  – OMT
  – Coad & Yourdon
  – BON
  – Shlaer & Mellor
  – Booch

• More specifically for real-time systems:
  – SDL
  – ROOM

… Quite a mess
What is UML?

UML = Unified Modeling Language

- Goal: lingua franca in modeling
- Standardized by the Object Management Group (OMG), a not-for-profit consortium of industry, academia, gov.
  - Major version: UML 2.0 (July 2005)
  - Current standard: UML 2.2 (February 2009)
- Definition driven by consensus rather than innovation
- Language definition style:
  - abstract syntax (meta-model)
  - static semantics (well formedness rules in OCL)
  - dynamic semantics (textual, in natural language)
  - concrete graphical syntax
  - usage notes
### Overview of the 14 diagrams of UML

<table>
<thead>
<tr>
<th>Structure diagrams</th>
<th>Behavior diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Class diagram</td>
<td>8. Use-case diagram</td>
</tr>
<tr>
<td>2. Composite structure diagram (*)</td>
<td>9. State machine diagram</td>
</tr>
<tr>
<td>3. Component diagram</td>
<td>10. Activity diagram</td>
</tr>
<tr>
<td>4. Deployment diagram</td>
<td><em>Interaction diagrams</em></td>
</tr>
<tr>
<td>5. Object diagram</td>
<td>11. Sequence diagram</td>
</tr>
<tr>
<td>6. Package diagram</td>
<td>12. Communication diagram</td>
</tr>
<tr>
<td>7. Profile diagram</td>
<td>13. Interaction overview diagram (*)</td>
</tr>
<tr>
<td></td>
<td>14. Timing diagram (*)</td>
</tr>
</tbody>
</table>

(*) not existing in UML 1.x, added in UML 2.0
UML principle: diagram vs. model

- Different diagrams describe various facets of the model
- Each diagram shows a projection of the model
- Several diagrams of the same kind may coexist
- Incoherence between diagrams (of the same or of different kind(s)) correspond to an ill-formed model
- The coherence rules between different kinds of diagrams is not fully stated
The four-layer metamodel hierarchy

(Example from UML Infrastructure formal/09-02-04, page 19)
Model-driven engineering

- Overview of UML
- System architecture, object structures
- Reactive behavior
- UML for real-time systems

About model driven engineering & embedded systems
UML : a family of languages
OMEGA UML
Timed simulation and model checking with IF
A case study
About models a
<table>
<thead>
<tr>
<th>Overview of UML</th>
<th>System architecture, object structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reactive behavior</td>
<td>UML for real-time systems</td>
</tr>
</tbody>
</table>

**UML : a family of languages**

- Overview of UML
- **System architecture, object structures**
- Reactive behavior
- UML for real-time systems
### Overview of the 14 diagrams of UML

<table>
<thead>
<tr>
<th>Structure diagrams</th>
<th>Behavior diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Class diagram</td>
<td>8. Use-case diagram</td>
</tr>
<tr>
<td>2. Composite structure diagram (*)</td>
<td>9. State machine diagram</td>
</tr>
<tr>
<td>3. Component diagram</td>
<td>10. Activity diagram</td>
</tr>
<tr>
<td>4. Deployment diagram</td>
<td><strong>Interaction diagrams</strong></td>
</tr>
<tr>
<td>5. Object diagram</td>
<td>11. Sequence diagram</td>
</tr>
<tr>
<td>6. Package diagram</td>
<td>12. Communication diagram</td>
</tr>
<tr>
<td>7. Profile diagram</td>
<td>13. Interaction overview diagram (*)</td>
</tr>
<tr>
<td></td>
<td>14. Timing diagram (*)</td>
</tr>
</tbody>
</table>

(*) not existing in UML 1.x, added in UML 2.0
Class diagrams: object structures

• The most known and the most used UML diagram

• Gives information on model structural elements

• Main concepts involved
  – Class - Object
  – Inheritance
  – Association (various kinds of)
Let’s start with … object orientation

• Why OO?
  – In the first versions, UML was described as addressing the needs of modeling systems in a OO manner
  – Statement not any longer maintained, however the OO inspiration for some key concepts is still there

• Main concepts:
  – **Object** – individual unit capable of *receiving/sending messages*, processing data
  – **Class** – pattern giving an abstraction for a set of objects
  – **Inheritance** – technique for reusability and extendibility

Further reading:
UML Class

- Gives the type of a set of objects existing at run-time
- Declares a collection of methods and attributes that describe the structure and behavior of its objects
- Basic notation:

<table>
<thead>
<tr>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>+serial_no : int</td>
</tr>
<tr>
<td>+mode : int</td>
</tr>
<tr>
<td>+pollData() : SensorData</td>
</tr>
</tbody>
</table>

- May own behavior (state machines, interactions, …)
- May be instantiated
  - except for abstract classes that can NOT be directly instantiated and exist only for inheritance
Characteristics of class features

- **Signature** (name, type)
- **Visibility** (public, private, protected, package)
- **Changeability** (changeable, frozen, addOnly)
- **Owner scope** (class, instance)

Additionally, **operations** are also characterized by
  - *pre* or *post* conditions
  - *body* (state machine or action description)
  - concurrency kind
Graphic syntax

Class name in italics: \textit{abstract} class

Underlined attribute = class attribute

Feature visibility \(+, -, #, ~\)

Simple class box: passive class

Overview of UML
System architecture, object structures
Reactive behavior
UML for real-time systems

About model driven engineering & embedded systems
UML: a family of languages
OMEGA UML
Timed simulation and model checking with IF
A case study
About models a
Inheritance

- A.k.a. generalization (specialization)
- Applies mainly on classes
- Other UML model elements can be subject to inheritance (e.g. interface)
  (if you want the exact list go check the UML metamodel for kinds of GeneralizableElements)
- Allows for polymorphism

```
Animal
  cry()

Cow
  cry(){moom}

Cat
  cry(){meow}
```
Association

- Concept with **no direct equivalent** in common programming languages
- Is defined as a **semantic relationship** between classes, that can materialize at runtime
- The *instance* of an association is a *set of tuples relating instances* of the classes
- It’s actual nature may vary, in terms of code, they may correspond to
  - Attributes, pointers
  - Operations
  - Network link to remote object

<table>
<thead>
<tr>
<th>Monitor</th>
<th>+monitor</th>
<th>+sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0..*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>+serial_no : int</td>
</tr>
<tr>
<td>+mode : int</td>
</tr>
<tr>
<td>+pollData() : SensorData</td>
</tr>
</tbody>
</table>
Association

- Association end
  - Endpoint of an association
  - Characterized by a set of properties contributing to the association definition
    - Multiplicity (ex: 1, 2..7, *, 4..*)
    - Ordering ordered/unordered
    - Visibility +,-,#, ~
    - Aggregation…

- Kinds of associations
  - Regular association
  - Composition: one class *is owned (composed in)* the associated class (implies *lifetime responsibility*)
Interface

- A UML classifier, similar to an abstract class
- Declares set of public features and obligations
- Specifies a contract, to be fulfilled by classes implementing the interface
- Not instantiable
- Its specification can be realized by classes
  - the class presents a public facade that conforms to the interface specification
  - a class may implement several interfaces
- Interfaces hierarchies can be defined through inheritance relationships
Interface definition and use examples

<<interface>>
InteractionPrimitives

tokenExchange()

<<interface>>
SecureInteraction

checkConsistency()
retrieveLast()

Satellite

SecureInteraction

The class Satellite provides implementation for the 3 operations
Communication primitives

- Communication is targeted to objects
- Signal
  - One way
  - Asynchronous communication primitive
  - May carry data parameters
  - It is defined independently of the classifiers handling it
- Operation call
  - One-way or two-way communication primitive (depending on signature)
  - Synchronous (the caller is blocked) or asynchronous
  - May carry data parameters
- Request management policy not constrained (e.g., existence of queues per instance, etc.)
Signal definition and use

- **Signal definition**
  - `<<signal>>` InitiateCall
    - calledNo : string

- **Interface**
  - `<<interface>>` PhoneConnection
    - `<<reception>>` InitiateCall

- **Class implementing the interface**
  - MobilePhone
    - PhoneConnection

- **Class able to receive a signal**
  - NetworkCell
    - `<<reception>>` InitiateCall

- Signal integrated in an **interface definition** via a Reception
Describing architecture: the composite structure diagram

- Added in UML 2.0
- Describes functional decomposition of systems/components
  - The internal structure of a classifier (sub-components)
  - Interaction points and links
- Concepts involved:
  - Classifier
  - Property
  - Port
  - Interface
  - Connection
Structured classifiers

- A class can have parts (i.e., objects owned by composition)

```
ATM

1 kb: Keypad

1 d: Display

1 ca: CardUnit

1 cu: CashUnit

1 cont: Controller

1 bb: BankTransactionBroker
```
Ports

- A class can have **ports typed with required and provided interfaces**

```
ATM

1 kb: Keypad

1 d: Display

1 ca: CardUnit

1 cu: CashUnit

1 cont: Controller

1 bb: BankTransactionBroker

ATMtoBank ATMtoBank

BankToATM

ATMMaintenance

ATM_Bank

BankToATM ATMtoBank

maintenance

Behavior port

Black-box view

Delegation port
```
Connectors

- Composite and components can be connected through links (connectors) – either directly or via ports
Connectors

- Composite and components can be connected through links (connectors) – either directly or via ports.
Black-box view

• shows the publicly visible properties defined through ports
• protocol state machine attached to ports may describe external view more precisely
Besides this…

- **Structured classifiers are like usual classes**
  - Usual **features**: attributes, associations, operations, state machine, inheritance
  - Structure is **inherited** and may be refined

- **Port types are not restricted**
  => e.g., they can own behavior. **What kind??**
...is largely left **unspecified** in the UML standard

**Some examples:**

```
Semantics
```

```
Overview of UML
System architecture, object structures
Reactive behavior
UML for real-time systems
```

```
About model driven engineering & embedded systems
UML : a family of languages
OMEGA UML
Timed simulation and model checking with IF
A case study
About models a
```

```
I. Ober - UML-based modeling and verification of real-time systems – Ecole Temps Réel, September 2009
```

```
```
Semantics

...is largely left **unspecified** in the UML standard

Some examples:
Semantics

…is largely left **unspecified** in the UML standard

Some examples:
Semantics

...is largely left **unspecified** in the UML standard

Some examples:
UML: a family of languages

- Overview of UML
- System architecture, object structures
- Reactive behavior
- UML for real-time systems
### Overview of the 14 diagrams of UML

<table>
<thead>
<tr>
<th>Structure diagrams</th>
<th>Behavior diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Class diagram</td>
<td>8. Use-case diagram</td>
</tr>
<tr>
<td>2. Composite structure diagram (*)</td>
<td>9. State machine diagram</td>
</tr>
<tr>
<td>3. Component diagram</td>
<td>10. Activity diagram</td>
</tr>
<tr>
<td>4. Deployment diagram</td>
<td>Interaction diagrams</td>
</tr>
<tr>
<td>5. Object diagram</td>
<td>11. Sequence diagram</td>
</tr>
<tr>
<td>6. Package diagram</td>
<td>12. Communication diagram</td>
</tr>
<tr>
<td>7. Profile diagram</td>
<td>13. Interaction overview diagram (*)</td>
</tr>
<tr>
<td></td>
<td>14. Timing diagram (*)</td>
</tr>
</tbody>
</table>

(*) not existing in UML 1.x, added in UML 2.0
Behavior modeling in UML

• **descriptive** (partial, declarative) behavior
  – use cases
  – interactions (sequence diagrams, collaborations)
  – high-level activity diagrams
  – logical constraints
  ⇒ no direct transformation into an implementation

• **prescriptive** (imperative, detailed) behavior
  – state machines
  – actions
  ⇒ design, close to implementation
State machines

• suitable for reactive behavior
  stimulus \Rightarrow response \Rightarrow state change

• basic idea: model system / component behavior as a finite automaton

• …plus many sophistications
  – describe parallel activities inside an object & synchronizations
  – factorize common transition behavior
States and transitions

* Model taken from the Rhapsody 5.2 samples
Regions, history

* Dishwasher model taken from the Rhapsody 7.4 samples
Other pseudostates

join / fork

final
entry point
exit point
Actions in state machines

- **transition effect**

- **state entry**

- **state exit**

- **state do activity**

```plaintext
ev
[x = 4]
/ x := 5
```

```
entry:
  print("s entered")

do
  ob.m()

exit:
  print("s exited")
```
Semantic variation points

VS.

/* no trigger here */
Actions

• No standard action language
  ⇒ no standard way to say \( o.m(7) \), or \( x := 9 \)

• A standard “action semantics” (since UML1.4)
  – defines the types of actions (with an abstract syntax):
    call, create object, create link, destroy object, raise exception,
    send object, send signal, ......................................................
  – but no concrete syntax (no keywords, punctuation,…)
  – only basic / no structured actions
    (structured actions obtained with activity graphs)
Actions in practice

in tools → a programming language + library / macros

drawbacks

- most actions do not exist in a programming language (send object, create/destroy link, erase attribute value, etc.)
- choice of programming language is irreversible
- model interchange is impossible
- over-specification
Concurrency model

• real-time systems are intrinsically concurrent
  – viewing different stimuli / requests as concurrent helps in respecting deadlines through scheduling

• in UML, several levels of concurrency
  – between objects
  – between operations of an object
  – between concurrent regions of a state machine

• concurrency model = how threads are defined / created /synchronized / stopped
  – Thread = abstract flow of control (≠ OS thread, may be mapped to)
Active / passive status

- specifies the *concurrency model* for classes
- specifies whether an *Object* instance of the *Class* maintains its own thread of control and runs concurrently with other active *Objects* (active)

```plaintext
Monitor
+monitor +sensors
1 0..*

Sensor
+serial_no : int
+mode : int
+pollData() : SensorData
```
Open issues

• concurrency model is not completely defined
  – can multiple active objects share a passive object?
  – how can the user control concurrent access to a passive object?
  – what is the postponing policy (queue, request table, etc.)?
  – what are the synchronization mechanisms available to designers?

• in practice, tools make particular assumptions (sometimes hidden)
  e.g. Rhapsody:
  – every passive object belongs to an active object
    ⇒ partition objects into activity groups
  – execution of an activity group is sequential
  – operation calls are only allowed inside a group
  – steps of different activity groups are interleaved…
UML: a family of languages

- Overview of UML
- System architecture, object structures
- Reactive behavior
- UML for real-time systems
Domain specific UML : profiles

UML can’t contain everything ! => extension mechanisms

- **Stereotypes** can be used to specialize any UML concept

Example:

**Stereotype definition**

```
<<metaclass>>
BehavioralFeature

<<stereotype>>
RealTimeFeature

occKind : ArrivalPattern
priority : NFP_Integer
```

**Stereotype use**

```
SpeedController

<<RealTimeFeature>> controlLoop()

occKind = periodic(period=(10,ms))
Priority = 42
```
Domain specific UML: profiles

Profile = a group of related and coherent stereotypes with associated well-formedness constraints and semantics (free)

- Packaged as a standalone UML model
- Can be applied to the domain-specific UML models
• **Standard UML 2.x profile for**
  **Modeling and Analysis of Real-time and Embedded systems**
  – In final phases of OMG standardization
  – First public version: 2007

• **Constructs for modeling:**
  – Non-functional properties
  – Different notions of **Time** (chronometric, logical, synchronous…)
  – **Resources**, resource allocation, policies, constraints
  – Particular component models and **models of computation**
  – Schedulability analysis and performance analysis information
  – ...

• Aims to be general, open, customizable, extensible
  => large quantity of predefined constructs
  => “generic” semantics (e.g., for the computation model)

• Adoption by tools under way
Overview

I. About model driven engineering & embedded systems
II. UML : a family of languages
III. OMEGA UML
IV. Timed simulation and model checking
V. A case study
What is OMEGA UML?

A large subset of UML\(^{(1)}\)  
+  
(More) model coherence constraints  
+  
A formal operational semantics\(^{(2)}\)  
+  
RT & Verification extensions\(^{(3)}\)

---

\(^{(1)}\) Structure (OO & components included), behavior (SM, actions)  
\(^{(2)}\) based on the Rhapsody tool semantics and defined in  
[Damm, Josko, Pnueli, Votintseva 2002 & Hooman, Zwaag 2003]  
\(^{(3)}\) Timing constraints, timed behavior (semantic projection to timed automata), property observers
Object structures

UML class diagrams

- active / passive classes
- associations
- composition
- generalization
Architecture
(since OMEGA2)

Relevant UML 2.0 composite structures
• (unidirectional) ports
• required/provided interfaces
• connectors, routing
• coherence rules
- state machines
- “primitive” operations
- imperative action language
  - assignments
  - control structure
  - communication
  - object creation
- communication:
  - asynchronous signals
  - asynchronous calls
  - synchronous blocking calls

• state machines
• “primitive” operations
• imperative action language
  – assignments
  – control structure
  – communication
  – object creation
• communication:
  – asynchronous signals
  – asynchronous calls
  – synchronous blocking calls

About model driven engineering & embedded systems

UML: a family of languages

OMEGA UML

Timed simulation and model checking with IF

A case study

About models
Composition & communication semantics

passive objs.

active obj.

activity groups (run-to-completion)

synchronous calls

asynchronous calls & messages

shared protected object (since OMEGA2)
formalizing requirements: observers

- special objects monitoring the system state & events and giving verdicts

**observer stereotype**

```uml
<<Observer>>
liftoff_performed_right2
```

**observer variables**

```uml
| g : Ground |
| mc : MissionConstants |
| tc : TimeConstants |
```

**wait_start**

```uml
match send ::EADS::Signals::Start(void) by g / begin mc := g.Acyclic.MissionConstants; tc := g.Acyclic.TimeConstants end
```

**wait_ignition_p1**

```uml
[ g.Acyclic.EAP.Pyro1 @ Ignition_done ]
[ now >= (tc.MN_5 * 2 + mc.Tpstar_prep) ]
[ now >= (tc.MN_5 * 2 + mc.Tpstot_prep) ]
```

**p1_ignited**

```uml
[ g.Acyclic.EAP.Pyro2 @ Ignition_done ]
[ now < (tc.MN_5*2 + mc.Tpstot_prep) ]
```

**choice**

```uml
<<error>>
ko
```

**error state**

```uml
ok
```

**event observation (see time profile)**

**state observation (variables + control states of reachable objects)**
Real-time

- **imperative** constructs: describe *time dependent behavior*
  - standard UML 2.0 time-related types: Time, Duration
  - mechanisms for measuring durations: timers, clocks
  - 0-ary operator `now : Time`

- **declarative** constructs: timed events and constraints
  orthogonal to the functional specification
  - separation of modeling concerns
  - flexible semantics: independent from the functional part
  - timed events: history of occurrence times of identified state changes
  - constraints (invariants) on duration between event occurrences
    - Assumptions (taken as hypotheses)
    - Requirements (to be verified)
Overview

I. About model driven engineering & embedded systems
II. UML : a family of languages
III. OMEGA UML
IV. Timed simulation and model checking
V. A case study
OMEGA toolset

**Goals:** early model validation & debugging

**Techniques:** timed automata simulation, model checking

**Principles:**
- Validation is an expensive task
  => **reuse** existing state-the-art (IF toolset)
- **Use standards** (UML profiling, XMI)
- **Be open** (to any UML editor if possible)
Architecture & functionality

- **Simulation**
  - Interactive, random,
  - Replay/analyze diagnostics...

- **Verification**
  - Observers, μ-calculus,
  - State graph minimisation
    (bisimulation),...

- **Static analysis**
  - Dead variable/code elimination, slicing,...
The IF language

Describes networks of communicating extended timed automata (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),

process (N1 initial instances)

s1(t1)

P1(N1)

s2 (t1, t2)
sr(1)

P3(N3)

signal
parameter

local data

I. Ober - UML-based modeling and verification of real-time systems – Ecole Temps Réel, September 2009
Process description

Process = hierarchical timed automaton

local data + local clocks

Principles
IF language and toolset
OMEGA UML
Timed simulation and model checking with IF
A case study
About models a

About model driven engineering & embedded systems
UML: a family of languages
OMEGA UML

process P1(N1);

// types, variables, constants, procedures
parameters
local data

state s0 ...;

... // transition t1

endstate;

state s1 #unstable...;

... // transitions t2, t3

endstate;

... // states s2, s3, s4

endprocess;

// states s2, s3, s4

P1(N1)
Transitions

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

\[
\text{state } s_0
\]

\[
\begin{align*}
\text{urgency} & \quad \text{eager} \\
\text{provided } x! &= 10; \\
\text{when } c_2 &\geq 4; \\
\text{input} &\quad \text{update}(m); \\
\text{body} &\quad \ldots \\
\text{nextstate} &\quad s_1;
\end{align*}
\]

\[
\begin{align*}
\text{untimed guard} \\
\text{timed guard} \\
\text{signal consumption} \\
\text{from the process buffer}
\end{align*}
\]

\[
= \text{trigger}
\]

\[
\text{statement} = \text{data assignment} \\
\text{message emission,} \\
\text{process or signalroute creation or destruction,} \\
\ldots
\]
Dynamic priorities

- priority order between process instances $p_1$, $p_2$ (free variables ranging over the active process set)

$$\text{priority\_rule\_name : } p_1 < p_2 \text{ if condition}(p_1,p_2)$$

- semantics: only maximal enabled processes can execute

- scheduling policies
  - fixed priority: $p_1 < p_2$ if $p_1$ instanceof T and $p_2$ instanceof R
  - run-to-completion: $p_1 < p_2$ if $p_2 = \text{manager}(0).\text{running}$
  - EDF: $p_1 < p_2$ if $\text{Task}(p_2).\text{timer} < \text{Task}(p_1).\text{timer}$
Compilation of OMEGA UML

• structure
  – UML object → IF process
  – attributes & associations → variables
  – inheritance: replication of structural features
  – composite structures → flat object network
    (ports → processes, connectors → variables)

• 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…)

Architecture & functionality

simulation
interactive, random,
replay/analyze diagnostics…

verification
observers, μ-calculus,
state graph minimisation
(bisimulation),…

static analysis
dead variable/code
elimination, slicing,…
Overview

I. About model driven engineering & embedded systems
II. UML: a family of languages
III. OMEGA UML
IV. Timed simulation and model checking
V. A case study
MARS overall architecture
Model architecture

```
MARS system

: DatabusManager
  AltMsgTimeoutCount : int
  NavMsgTimeoutCount : int
  altDataTimer : Timer
  navDataTimer : Timer

: ControllerMonitor
  currentStatus : int
  previousStatus : int

Environment

navDataSource : DataSource
  cOffset : Clock
  tPeriod : Timer
  cJitter : Clock

altDataSource : DataSource
  cOffset : Clock
  tPeriod : Timer
  cJitter : Clock

: DatabusController
  status : int
```
Data bus manager: timing requirements
Environment model: data sources

```
Init

/cOffset.set(0)

[cOffset <= 25] / tPeriod.set(25)

WaitCycle

/cJitter <= 10
/ self.sendData()

timeout(tPeriod) / begin
cJitter.set(0);
tPeriod.set(25)
end

ProduceData

[cJitter <= 10]
/ informal "lost data"
```
Property: measuring reactivity R1

Absence of transmission is discovered within delay BR1

<<observer>>
PR1

BR1: duration
c: Clock

I. Ober - UML-based modeling and verification of real-time systems – Ecole Temps Réel, September 2009
Model of the Databus Manager
Abstract model, Version 1: for every sender

- Determine for every period “signal ok” or “signal loss” and count them
  - in state operational: count consecutive losses
  - in state BusError: count consecutive receptions
- Depending on counter values decide if status switch is needed

Verification shows: reactivity property holds
Version 2: Asymmetric

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

Verification shows: this version is less reactive
## Verification parameters and results

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Number of states</th>
<th>Number of transitions</th>
<th>User time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial model with only one source (no CM polling)</td>
<td>1084</td>
<td>1420</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>(non-conservative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial model with two synchronized sources (no CM polling)</td>
<td>99355</td>
<td>151926</td>
<td>36s</td>
</tr>
<tr>
<td>(non-conservative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial model with two de-synchronized sources (no CM polling) (conservative – does not finish)</td>
<td>&gt; 1136768</td>
<td>&gt; 1676126</td>
<td>&gt; 9m30s</td>
</tr>
<tr>
<td>Abstract model, 10ms CM polling (conservative – does not finish)</td>
<td>&gt; 1494864</td>
<td>&gt; 701120</td>
<td>&gt; 8m12</td>
</tr>
<tr>
<td>Abstract model with no CM polling (non-conservative)</td>
<td>118690</td>
<td>174871</td>
<td>45s</td>
</tr>
<tr>
<td>Abstract model with lazy CM polling (conservative)</td>
<td>155166</td>
<td>263368</td>
<td>1m21s</td>
</tr>
</tbody>
</table>
compositional re-design
⇒ better abstraction results

- Error logic parametric in the number of receivers
- Project property to a single receiver
- Consider 1 concrete receiver and abstract all others
Conclusions

• Push button verification for whole systems – not there yet!

• Still useful for analysis of "hard points"

• Essential success factors:
  – "user-friendly" language and tools
  – use of abstractions
    • easy ones (∼ automatic): timing relaxation, …
    • harder ones: design refactoring ← "design for verification" skills
Wrap-up

- UML: standard language with fairly advanced architecture & behavior features
- Well established in the RT domain
- Used for:
  - System-level specification (systems of systems, etc.)
  - Detailed functional design
  - Low-level architecture (concurrency, deployment,…)
  - Quantitative analysis (scheduability, performance,…)
  - ??
- Open semantic issues, analysis problems
  => still much work to do for PhD students