The IF toolset

VERIMAG


4th International School on
Formal Methods for the Design of
Computer, Communication and Software Systems:
Real Time

September 2004
The IF toolset: 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 toolset: approach

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

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

Optimisation and abstraction

Transition systems

simulation

test

verification1

verification2

verification3

state explosion
The IF toolset: challenges for IF

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
Outline

Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion
Thesis:
A Timed Model of a RT system can be obtained by “composing” its application SW with timing constraints induced by both its execution and its external environment.
## Modeling real-time systems

<table>
<thead>
<tr>
<th></th>
<th>Application SW</th>
<th>Timed model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DESCRIPTION</strong></td>
<td>Reactive machine (untimed)</td>
<td>Reactive machine + External Environment + Execution Platform</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>Reference to physical (external) time</td>
<td>Quantitative (internal) time Consistency pbs- timelocks</td>
</tr>
<tr>
<td><strong>TRIGGERING</strong></td>
<td>Timeouts to control waiting times</td>
<td>Timing constraints on interactions</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>TO(5) <img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /> <img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>ACTIONS</strong></td>
<td>No assumption about Execution Times Platform-independent</td>
<td>Assumptions about Execution Times Platform-dependent</td>
</tr>
</tbody>
</table>
Modeling real-time systems – Taxys (1)

Environment

Deadline constraint:
\[ t_{\text{out}} - t_{\text{in}} < d \]

Throughput constraint:
no buffer overflow
Modeling real-time systems – Taxys (2)

- **ESTEREL + C Data**
- **SAXO-RT**
- **C Code**
- **SAXO**
- **Machine Description**
- **Target Machine executable code**
- **Event Handler Timed Model**
- **Environment Timed Model**
- **C2TimedC**
- **Timed (instrumented) C Code**
- **Exec. Times**
- **Timing Diagnostics**
- **IF/KRONOS**
Modeling real-time systems – Taxys(3)

Application = ESTEREL + Pragmas

Environment = ESTEREL + Pragmas

Event Handler

Instrumented C Code

Target Machine Executable Code

QoS requ.

Exec.T

SAXO-RT

Instrumented C Code

KRONOS Algorithms and Data Structures

IF/KRONOS

Timing Diagnostics
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
From application SW to implementations

Environment

Application software

Task1
Event handler
Resource management and Task synchronization

Task2

Task3

Task4

Scheduler

Platform
From application SW to implementations

- Lustre
- Esterel
- ADA
- SDL
- RT- Java
- UML
- C
- C++
- Jini
- CORBA
- TTA
- CAN
- RTOS
- OSEK
- DSP
- \( \mu \)controller
From application SW to implementations

**Functional, Logical, Abstract time,**
*High level structuring constructs and primitives*
*Simplifying synchrony assumptions wrt environment*

**Abstraction**

**Implementation**

**Physical, Non functional properties**
*Execution times, interaction delays, latency, QoS*
*Mapping functional design into tasks, data, resources*
*Task coordination, resource management, scheduling*
From application SW to implementations – synchronous vs. asynchronous

**Synchronous**
- Lustre, Esterel
- Statecharts
- Non interruptible execution steps
- Usually, single task, single processor
- "Everybody gets something"

**Asynchronous**
- ADA, SDL
- Event triggered
- Multi-tasking
  - RTOS
- Usually static Priorities – RMA
- "Winner takes all"

**Application SW**
**Component based approaches**

**Implementation**
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Component-based construction

Build systems by **composition of components**

**Component** = 
Interface (set of interactions) + Behavior (transition system)

**Composition operation** allows building new components
Component-based construction

Construction problem:
Given a component $C$ and a property $P$ find $C'$ and $\parallel$ such that $C \parallel C'$ satisfies $P$

Composition:
- Creates new interactions
- Restricts the behavior of the components

Key issue: Heterogeneity
Interactions are specified by connectors. They can be
• strict (rendez-vous in CSP) or non strict (msg sending,broadcast)
• atomic (rendez-vous) or non atomic (asynchronous comm.)
• binary (point to point as in CCS, SDL) or n-ary in general
Composition - restriction

Restrictions enforce properties of execution such as synchrony, scheduling policies, run-to-completion.

Synchronous execution is a restriction of asynchronous execution
Composition - heterogeneity of interaction and execution

A: Atomic interaction                S: Strict interaction

Asynchronous Execution

Synchronous Execution

Lotos
CSP

Java
UML

SDL
UML

A S
nonA S
A nonS
nonA nonS

Esterel, Lustre
VHDL
Statecharts1

Statecharts2
Composition: incrementality

Use a unique binary associative compositon operation (express n-ary composition by binary composition)
Outline

Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion
Layered system construction

A **component** is a pair \((B, IM)\) where

- **\(B\)** *is a transition system*
- **\(IM\)** *an interaction model*

Composition operators:

- *Parallel composition*: \((B_1, IM_1) \parallel_{IM[1,2]} (B_2, IM_2) = (B, IM)\)
- *Restriction to enforce a property \(p\)*: \((B, IM) \rightarrow (\parallel p, IM)\)
Parallel composition: Interaction models - examples

NB: Only complete or maximal incomplete interactions are legal!
Parallel composition: Interaction models - definition

Let $K$ is a set of component names with disjoint action vocabularies $A_i$ for $i \in K$.

A connector $c$ of $K$ is a non empty subset of $\cup_{i \in K} A_i$ such that $|c \cap A_i| \leq 1$

The set of the interactions of a connector $c$, $I(c)$, is the set of all the non empty subsets of $c$.

An interaction model IM is a pair $IM=(C, I(C)+)$

- A set of connectors $C$ or equivalently the set of the interactions of $C$, $I(C) = \cup_{c \in C} I(c)$

- A set of the complete interactions $I(C)+$, $I(C)+ \subseteq I(C)$ such that $a \in I(C)+$, $a \subseteq a'$ implies $a' \in I(C)+$
Parallel composition: Interaction models - composition

**IM[K₁,K₂]:**

C[K₁,K₂] = {{a₁, a₂, a₃, a₄}, {a₁₁, a₁₂}}

IC[K₁,K₂]⁺ = {a₁|a₂|a₃|a₄, a₁₁, a₁₁|a₁₂}

**IM[K₁]:**

C[K₁] = {{a₁, a₂}, {a₅, a₉},{a₆, a₉}}

IC[K₁]⁺ = {a₅, a₆, a₁₁, a₅|a₉, a₆|a₉}

**IM[K₂]:**

C[K₂] = {{a₃, a₄}, {a₇, a₁₀}, {a₈, a₁₀}}

IC[K₂]⁺ = {a₁₀, a₇|a₁₀, a₈|a₁₀}
Parallel composition: Interaction models – composition (2)

\[
\text{IM}[K_1, K_2]: \\
C[K_1, K_2] = \{\{a_1, a_2, a_3, a_4\}, \{a_{11}, a_{12}\}\} \\
\text{IC}[K_1, K_2]^+ = \{a_1|a_2|a_3|a_4, a_{11}, a_{11}|a_{12}\}
\]

\[
\begin{align*}
\text{IM}[K_1]: \\
C[K_1] &= \{\{a_1, a_2\}, \{a_5, a_9\}, \{a_6, a_9\}\} \\
\text{IC}[K_1]^+ &= \{a_5, a_6, a_{11}, a_5|a_9, a_6|a_9\}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{K}_1 & a_1 & a_2 \\
& a_5 & a_6 \\
& a_9 & a_{11}
\end{array}
\]

\[
\begin{array}{ccc}
\text{K}_2 & a_3 & a_4 \\
& a_7 & a_8 \\
& a_{10} & a_{12}
\end{array}
\]

\[
\begin{align*}
\text{IM}[K_2]: \\
C[K_2] &= \{\{a_3, a_4\}, \{a_7, a_{10}\}, \{a_8, a_{10}\}\} \\
\text{IC}[K_2]^+ &= \{a_{10}, a_7|a_{10}, a_8|a_{10}\}
\end{align*}
\]

\[
\begin{align*}
\text{IM}[K_1 \cup K_2]: \\
C[K_1 \cup K_2] &= C[K_1] \cup C[K_2] \cup C[K_1, K_2] \\
\text{IC}[K_1 \cup K_2]^+ &= \text{IC}[K_1]^+ \cup \text{IC}[K_2]^+ \cup \text{IC}[K_1, K_2]^+
\end{align*}
\]

\[
\begin{array}{ccc}
\text{K}_1 \cup \text{K}_2 & a_1 & a_2 \\
& a_5 & a_6 \\
& a_9 & a_{11}
\end{array}
\]
Parallel composition: General definition

\[ IM[K_1, K_2] \]

\[ IM[K_1] \]
\[ B[K_1] \]
\[ S[K_1] \]

\[ IM[K_2] \]
\[ B[K_2] \]

\[ IM[K_1 \cup K_2] \]
\[ B[K_1 \cup K_2] \]
\[ S[K_1 \cup K_2] \]

\[ S[K_1] \parallel S[K_2] = (B[K_1], IM[K_1]) \parallel (B[K_2], IM[K_2]) \]
\[ = (B[K_1] \times B[K_2], IM[K_1] \cup IM[K_2] \cup IM[K_1,K_2]) \]
\[ = S[K_1 \cup K_2] \]

where \( \times \) is an associative and commutative operation such that

\[ B[K_1] \times B[K_2] = B[K_1 \cup K_2] \]

Composition is associative and commutative
Flexible parallel composition: transition systems with priorities

**Behavior**: transition systems

**Interaction model**: priority relation on interactions

A transition system with priorities is a pair \((B, \prec)\) where,
- \(B\) is a labeled transition system with labels from a set of interactions \(A\)
- \(\prec\) is a strict partial order on \(A\) that restricts \(B\):

**Semantics of \((B, \prec)\)**:

\[ q \xrightarrow{a_1} q' \in (B, \prec) \text{ if } q \xrightarrow{a_1} q' \in B \text{ and there is no } q \xrightarrow{a_2} q'' \in B, \ a_1 \prec a_2 \]

The sum \(\prec^1 \oplus \prec^2\) of two priority orders \(\prec^1, \prec^2\) is the least priority order (if it exists) such that \(\prec^1 \cup \prec^2 \subseteq \prec^1 \oplus \prec^2\)

**Remark**: \(\oplus\) is a (partial) associative and commutative operation
Flexible parallel composition - definition

Composition of behaviors:

\[ q_1 \xrightarrow{a_1} q_1', \quad q_2 \xrightarrow{a_2} q_2' \] implies \[ \begin{cases} (q_1, q_2) \xrightarrow{a_1} (q_1', q_2) \\ (q_1, q_2) \xrightarrow{a_2} (q_1, q_2') \\ (q_1, q_2) \xrightarrow{a_1} a_2 \rightarrow (q_1', q_2') \text{ if } a_1 \mid a_2 \in IC[K_1 \cup K_2] \end{cases} \]

\[ \langle^{12} \rangle \text{ is defined by the rules:} \]

- **Maximal progress:** \( a_1 \langle^{12} \rangle a_1 \mid a_2, \) if \( a_1 \mid a_2 \in IC[K_1 \cup K_2] \)
- **Completeness:** \( a_1 \langle^{12} \rangle a_2, \) if \( a_1 \) is incomplete and non-maximal \( a_2 \) is complete in \( IC[K_1 \cup K_2] \)
Flexible parallel composition: producer-consumer

Producer

Producer

Consumer

Consumer

put

get

put

get

put

get

put

get

prod

cons

put | get

prod

put

get

cons

Producer || Consumer
Flexible parallel composition: deadlock-freedom by construction

\[(B^1, \langle 1 \rangle) \parallel (B^2, \langle 2 \rangle) = (B^1 \times B^2, \langle 1 \oplus 2 \oplus 12 \rangle)\]

is an associative total operation on components if no incomplete interaction dominates a complete interaction in the components.

\[(B, \langle \rangle)\] is deadlock-free if \(B\) is deadlock-free.

\[(B^1, \langle 1 \rangle) \parallel (B^2, \langle 2 \rangle)\] is deadlock-free if \(B^1, B^2\) are deadlock-free.

*Check that after composition the resulting component cannot execute incomplete interactions which are not maximal.*
Outline

Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion
Adding timing constraints

- There exist different timed extensions for $P_1 \parallel P_2$ corresponding to different assumptions about idling before interaction.

- Compositionality: define $P_1^T \parallel P_2^T$ so as to preserve properties such as well-timedness, deadlock-freedom, liveness.
Adding timing constraints: Timed systems

Automata: labeled transition relations on a set of actions

Timers: real-valued variables that can
- be reset and tested at transitions
- increase (derivative =1) or remain unchanged at states (derivative =0)

Types of urgency $\tau$ associated with guards express priority over time progress at states

$\varepsilon$ (eager): if enabled then must fire asap
$\lambda$ (lazy): if enabled then may fire
$\delta$ (delayable): if enabled must fire before it becomes disabled
Adding timing constraints: example

A periodic process of period $T$ and execution time $E$

Actions
- $a$: arrive
- $b$: begin
- $e$: end

$t' = x' = 1$ at all states
Adding timing constraints

Three different kinds of timing constraints:

• *from the execution platform* e.g. execution times, latency times

• *from the external environment* about arrival times of triggering events e.g. periodic tasks

• *user requirements* e.g. QoS, which are timing constraints relating events of the real-time system and events of its environment e.g. deadlines, jitter
Adding timing constraints

Each shared resource induces a partition on the control states of a process \{Sleep, Wait, Use\}.

\[
\begin{align*}
\text{Sleep} & \quad \text{Arrival times (t)} \quad t := 0 \quad T_{\min} \leq t \leq T_{\max} \\
\text{Wait} & \quad \text{Execution times (x)} \quad x := 0 \quad (E_{\min} \leq x \leq E_{\max}) \\
\text{Use} & \quad \text{Deadline D} \quad t \leq D - E_{\max} \quad (t \leq D)
\end{align*}
\]
Outline

Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion
A scheduler is a controller which restricts access to resources so as to meet the timing constraints (deadlock-free behavior) by applying a scheduling policy $K_{pol}$:

$$K_{pol} = \bigwedge_{r \in R} K_{r_pol}$$

$$K_{r_pol} = K_{r_res} \land K_{r_adm}$$

$K_{r_res}$ says how conflicts for the acquisition of resource $r$ are resolved e.g. EDF, RMS, LLF

$K_{r_adm}$ says which requests for $r$ are considered by the scheduler at a state e.g. masking
Scheduler modeling

Example: $K_{pol}$ for the Priority Ceiling Protocol

Admission control: “Process $P$ is eligible for resource $r$ if the current priority of $P$ is higher than the ceiling priority of any resource allocated to a process other than $P$”

Conflict resolution: “The CPU is allocated to the process with the highest current priority”

Result: Any feasible scheduling policy $K_{pol}$ induces a restriction that can be described by dynamic priorities
Outline

Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion
Timed Systems with priorities

Priority rule | Strengthened guard of bgn1
---|---
true → bgn1 ∩ bgn2 | \( g_1' = g_1 \wedge \neg g_2 \)
C → bgn1 ∩ bgn2 | \( g_1' = g_1 \wedge \neg (C \wedge g_2) \)
Timed Systems with priorities

A priority order is a strict partial order, \( \langle \subseteq \mathcal{A} \times \mathcal{A} \rangle \)

A set of priority rules, \( \text{pr} = \{ C_i \rightarrow \langle_i \} \) where \( \{C_i \} \) is a set of disjoint state predicates

\[
g'_k = g_k \land \bigwedge C \rightarrow \langle \in \text{pr} \left( C \Rightarrow \bigwedge a_k \langle a_i \rightarrow g_i \right) \right) \]
Timed Systems with priorities: FIFO policy

\[ t_1 \leq t_2 \rightarrow b_1 \prec b_2 \quad t_2 \leq t_1 \rightarrow b_2 \prec b_1 \]
Timed Systems with priorities: Least Laxity First policy

\[ L_1 \leq L_2 \rightarrow b_2 \preceq b_1 \quad L_2 \leq L_1 \rightarrow b_1 \preceq b_2 \]

where \( L_i = T_i - E_i - t_i \) is the laxity of process \( i \)

```
\begin{align*}
\text{sleep1} & \quad \text{a1} \\
& \text{t1=T1} \\
& \text{t1:=0} \\
\text{wait1} & \quad \text{b1} \\
& \text{t1\leq T1-E1} \\
& \text{x1:=0} \\
\text{use1} & \quad \# \\
\text{use2} & \quad \# \\
\end{align*}
```
Timed Systems with priorities: composition of priorities

\[(pr_1 \oplus pr_2)(q)\] is the least priority order containing \[pr_1(q) \cup pr_2(q)\]

Results:
- The operation \(\oplus\) is partial, associative and commutative
- Sufficient conditions for deadlock-freedom and liveness
### Timed Systems with priorities: mutual exclusion + FIFO

<table>
<thead>
<tr>
<th>Condition</th>
<th>Transition 1</th>
<th>Transition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 \leq t_2 )</td>
<td>( b_1 \prec b_2 )</td>
<td>( b_2 \prec b_1 )</td>
</tr>
<tr>
<td>( t_2 \leq t_1 )</td>
<td>( b_2 \prec b_1 )</td>
<td>( b_2 \prec b_1 )</td>
</tr>
<tr>
<td>( \text{true} )</td>
<td>( b_1 \prec e_2 )</td>
<td>( e_2 )</td>
</tr>
</tbody>
</table>
Systems with priorities: Fixed priority preemptive scheduling

### Scheduling policy

\[ b_i < b_j, \quad b_i | p_k < b_j | p_k, \quad f_i | r_k < f_j | r_k \quad \text{for} \quad n \geq i > j \geq 1 \]

### Interaction model

\[ b_j | p_i, \quad f_j | r_i \in IC, \quad \text{for} \quad n \geq i, j \geq 1 \]

\[ a_i, f_i, b_i \in IC^+, \quad \text{for} \quad n \geq i \geq 1 \]
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
IF notation: 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
IF notation: System description

• 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
const N1 = ...;  // constants

N1 = ...;

// constants

type t1 = ...;  // types

t1 = ...;

// types

signal s2(t1, t2),  // signals

// signals

// signalroutes

signalroute sr1(1) ...  // route attributes from P1 to P3

// signalroute sr1(1) ...

// route attributes from P1 to P3

// processes

process P1(N0)

...  // data +

behaviour

data +

endprocess;

... process P3(N3)

...  // data +

endprocess;

process P3(N3)
IF notation: Process description

Process = hierarchical, timed systems with actions

```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;
```

Parameters

Local data

Outgoing transitions

Local data + local clocks

P1(N1)
IF notation: dynamic creation

- **Process creation:**
  
  
  - $p := \text{fork client (true)}$
    
    - pid of the newly created instance
    
    - process name
    
    - a new instance is created

  - **Process destruction:**
    
    - kill client(2)
      
      - pid expression
      
      - the instance is destroyed, together with its buffer, and local data

    - kill $p$
      
      - pid expression

    - **Process termination:**
      
      - stop
      
      - the “self” instance is destroyed, together with its buffer, and local data
IF notation: Process description-transition

transition = \textit{urgency} + \textit{trigger} + \textit{body}

\textbf{statement} = \textit{data assignment, message emission, process or signalroute creation or destruction, ...}
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

signalroute s1(1) #unicast #lossy #fifo
from server to client with grant, fail;

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
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;`

Signal consumption (blocking):

`input req (f, s);`
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;
The model of time [timed systems]

- global time → same clock speed in all processes
- time progress in stable states only → transitions are instantaneous
IF notation: 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;

nextstate wait_ack;
endstate;

state wait_ack;
input ack(sender,c);
...

... endstate;
```
IF notation: dynamic priorities

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

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

• semantics: only maximal enabled processes can execute

• scheduling policies
  – fixed priority:
  – run-to-completion:
  – EDF:
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Core components

syntactic transformation tools:
- static analyser
- code generator

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

IF specifications
reader
writer

IF AST

C/C++ code

application specific process code
predefined modules
(time, channels, etc.)

asynchronous execution
dynamic scheduling

state space representation

compiler
Core components: syntactic transformations

- Gives programming access to the AST of an IF description
- AST represented as a collection of C++ objects

syntactic transformation tools:
- static analyser
- code generator
Core components: exploration platform - API

- gives programming access to the underlying labeled transition system of an IF description

- the API provides
  - state, label representation
    - type definition
    - access primitives
  - forward traversal primitives
    - initial state function (init)
    - successor function (post)

- on-the-fly, forward, explicit, enumerative

![Diagram showing the components of the API]

- application specific process code
- predefined modules (time, channels, etc.)
- asynchronous execution
- dynamic scheduling
- state space representation

LTS exploration tools
- debugging
- model checking
- test generation
Core components: exploration platform

Offers primitives for exhaustive state space exploration

Main features

– process execution simulation
  • inter-process interaction
  • process creation / destruction
  • control of simulation time
– non-determinism handling
  • asynchronous execution
  • internal non-deterministic choices
  • open environment
– state space representation
Core components: exploration platform - architecture

- Exploration API
- Dynamic scheduling
- Asynchronous execution
- State space representation
- C/C++ code
- Application specific process code
- Predefined modules (time, channels, etc.)
- Compiler
- IF AST
Core components: exploration platform – execution

1\textsuperscript{st} layer: emulates asynchronous parallel execution to obtain global (system) steps from local (process) steps

– it asks successively, each process instance to execute its enabled transitions
– during the execution of a transition by a process instance,
  • it ensures message delivery and shared variable update
  • it manages dynamic instance creation and destruction
  • it records generated observable events
– when a local step is finished,
  • It takes a snapshot of the global configuration and stores it
  • It sends the successor to the 2\textsuperscript{nd} layer (dynamic scheduler)
– It manages time progress and clocks updates
Core components: exploration platform – execution

2<sup>nd</sup> layer: dynamic scheduling (priorities)
- collects all potential global successors
- filters them according to dynamic priorities
  - evaluates each priority constraint
  - if applicable on current state, it removes successors produced by the low priority instance
- delivers the remaining set to the user application through the exploration API
Core components: exploration platform – execution

- **Active instances**: I₁:P₁, I₂:P₁, I₁:P₂, I₂:P₂, Iₖ:Pₖ
- **Execution control**: asynchronous execution
- **Dynamic scheduling**:
  - **Succ?**
  - **Succ!**
- **Time module**:
  - **Set, reset**
  - **Create**
  - **Run**
  - **Output**
  - **Step**
  - **Run**
  - **Step**
  - **Run**
  - **Step**

To summarize, the core components include active instances and execution control, which are part of a larger system involving time management, scheduling, and execution steps. The diagram illustrates the dynamic interactions and control mechanisms within the exploration platform.
Core components: exploration platform – 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
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Validation

Model-Based Validation
- model checking
- test generation
- optimization
- static analysis
**Validation: 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
[b = R(0).flag]

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

[x >= t_ack]
```
Validation: requirements - using $\mu$-calculus

- **alternating-free** fragment

  \[ \phi ::= T \mid X \mid <a>\phi \mid \neg\phi \mid \phi \land \phi \mid \mu X. \phi(X) \]

  where $a$ denotes a regular expression on labels

- **macros** available to describe complex formula e.g.,

  \[
  \text{all } \phi \equiv \nu X. \phi \land [\ast]X \\
  \text{pot } \phi \equiv \mu X. \phi \lor <>X \\
  \text{inv } \phi \equiv \mu X. \phi \lor <>T \land [\ast]X
  \]

- **On-the-fly local** model-checker

- **diagnostics** can be extracted either as **sequences** (if the property is “linear”) or **sub-graphs** (if the property is “branching”)
Validation: behavioral equivalence checking

• **LTS comparison:**
  – equivalence relations ("behavior equality"):  
    System \( \approx \) Requirements  
  – preorder relations ("behavior inclusion"):  
    System \( \leq \) Requirements

• **LTS minimization:**
  – quotient w.r.t an equivalence relation:  
    (System / \( \approx \))

• **CADP can be used to check the following relations:**  
  weak/strong bisimulation, branching, safety, trace equivalence
Validation: behavioral equivalence checking

reduction w.r.t. branching bisimulation
Validation: optimization

• User defined costs associated to transitions of IF descriptions e.g., execution times

• problem: find the min-cost execution path leading from some 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)
Validation: 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
Validation: static analysis – live variables

a variable is **dead** at a control point if its value is not used before being redefined on any path starting at that point

**find live variables**
usual backward dataflow analysis extended to IF interaction primitives
asynchronous interaction via queues
parameter passing at process creation
live variables are propagated both intra and inter processes!

**exploit live variables**
transform IF description by
removing completely dead variables and signal / process parameters
resetting partially dead variables

the gains are multiple:
drastically reduce the size of the model
(orders of magnitude on realistic examples)
strongly preserve the initial behaviour
Validation: static analysis – dead code elimination

A part of code is dead if it will never been entered, for any execution.

- **find dead code**
  - Algorithm for static accessibility of control states and control transitions given user assumptions about the environment.
  - Accessibility propagated both intra- and inter processes.

- **exploit dead code**
  - Transform IF description by removing processes never created, signals never sent, and unreachable control states and control transitions.
  - The gains are reduction in the size of the description, enable more reduction by live analysis, and strongly preserve the initial behavior, under the given assumptions.

Diagram:
- Process P(1) with actions ?a, ?b, and !c.
- Process Q(0) with actions ?b and fork R.
- Process R(0) with actions ?c and !a.

(a) provides only “a” signals to the process P.
Validation: static analysis – variable elimination

abstraction w.r.t. a set of variables (to eliminate) provided by the user

find undefined variables
forward dataflow analysis propagating the influence of removing variables
local undefined-ness of variables
global undefined-ness of signal and process parameters
the propagation is performed both intra- and inter-processes

exploit undefined variables
transform IF descriptions by
removing assignments to undefined variables
removing undefined signal and process parameters
relaxing guards involving undefined variables
obtain a conservative abstraction of the initial description i.e., including all the behaviors of the initial one
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Front-Ends
- sdl2if
- uml2if
UML for real-time and embedded systems (OMEGA IST project)

• covers operational specifications
  – classes with operations, attributes, associations, generalization, statecharts; basic data types

• defines a particular execution model
  – a notion of active class
  – instances of active classes define activity groups
  – run-to-completion for activity groups

• interaction and behavior
  – primitive operations – procedural, stacked
  – triggered operations – embedded in state machine, queued
  – asynchronous signals

• define an Action Language
Front ends: UML2IF – translation principle

• **structure**
  – class → process type
  – attributes & associations → variables
  – inheritance → replication of features
  – signals, basic data types → direct mapping

• **behavior**
  – state machines (with restrictions) → IF hierarchical automata
  – action language → IF actions, automaton encoding
  – operations:
    • operation call/return → signal exchange
    • procedure activations → process creation
    • polymorphism → untyped PIDs
    • dynamic binding → destination object automaton determines the executed procedure
Front ends: UML2IF – architecture

- Rhapsody
- Rose
- Argo
- Objecteering

UML2IF
- XMI reader
- IF 2.0 translator
- UML 1.4 repository
- UML 1.4 API

IF 2.0 TOOLBOX

IF description

XMI 1.0/1.1 (UML 1.4 + stereotypes)
Front ends: UML2IF – simulation interface

- user friendly simulation
- system state exploration...
- customizable presentation of results for UML users
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Case studies: protocols

**SSCOP**
Service Specific Connection Oriented Protocol

**MASCARA**
Mobile Access Scheme based on Contention and Reservation for ATM case study proposed in *VIRES ESPRIT LTR*

**PGM**
Pragmatic General Multicast case study proposed in *ADVANCE IST-1999-29082*
Case studies: distributed applications

TCP/ECN Transit Computerization Project
  case study proposed in AGEDIS IST-1999-20218

MQ Series Integration Broker
  case study proposed in AGEDIS IST-1999-20218
Case studies: manufacturing

Job-shop Scheduling

Axxom Lacquer Production

case study proposed in AMETIST IST-2001-35304
Case studies: asynchronous circuits

**timing analysis**

O. Maler et al. *On timing analysis of combinational circuits*. In *Proceedings of the 1st workshop on formal modeling and analysis of timed systems, FORMATS’03, Marseille, France*.

**functional validation**

D. Borrione et al. *Validation of asynchronous circuit specifications using IF/CADP*. In *Proceedings of IFIP Intl. Conference on VLSI, Darmstadt, Germany*
Case studies: Embedded software

Ariane 5 Flight Program

joint work with EADS Launchers

K9 Rover Executive

S. Tripakis et al. **Testing conformance of real-time software by automatic generation of observers.** In *Proceedings of Workshop on Runtime Verification, RV’04, Barcelona, Spain.*

Akhavan et al. **Experiment on Verification of a Planetary Rover Controller.** In *Proceedings of 4th International Workshop on Planning and Scheduling for Space, IWPSS’04, Darmstadt, Germany.*
Ariane-5 flight program
Flight program specification

• built by reverse engineering by EADS-LV

• two independent views
  1. asynchronous
     – high level, non-deterministic, abstracts the whole program as communicating extended finite-state machines
  2. synchronous
     – low level, deterministic, focus on specific components …

– we focus on the asynchronous view
Flight program architecture

OBC (On Board Computer)

- Regulation
  - engines/boosters
  - ignition/extinction

- Configuration
  - stage/payload
  - separation

Control
- Navigation
- Guidance
- Algorithms

Ground

OBC (Redundant)

~3500 lines of SDL code
Regulation components

- initiate **sequences** of “regulation” **commands** at right moments in time:
  - at $T_0 - \Delta_1$ execute action$_1$
  - at $T_0 - \Delta_2$ execute action$_2$
  - at $T_0 - \Delta_n$ execute action$_n$

- if necessary, stopped at any moment

- described as “**sequential**” processes, moving on specific, precise times
Configuration components

- initiate “configuration” changes depending on:
  - flight phase: ground, launch, orbit, ...
  - control information: reception of some signal, ...
  - time: eventually done in $[T_0+L, T_0+U]$

- described as processes combining signal and timeout-driven transitions
the opening action eventually happens between $T_{\text{early}}$ and $T_{\text{late}}$ moments, if possible, on the reception on the open signal.
Control components

• compute the flight commands depending on the current flight evolution
  – guidance, navigation and control algorithms

• abstracted over-simplified processes
  – send flight commands with some temporal uncertainty
Control components: example

time non-deterministic:
the firing signal can be sent between $T_0 + L$ and $T_0 + U$

- **init**
  - lazy
    - $T_0 + L \leq \text{now}$ and $\text{now} \leq T_0+U$
    - output firing to vulcain
  - done

time deterministic:
the firing signal is sent exactly at $T_0 + K$

- **init**
  - eager
    - $T_0 + K = \text{now}$
    - output firing to vulcain
  - done
Flight program requirements

• **general** requirements
  – e.g, no deadlock, no timelock

• **overall system** requirements
  – e.g, flight phase order
  – e.g, stop sequence order

• **local component** requirements
  – e.g, activation signals arrive eventually in some predefined time intervals
Validation: model exploration

- test simple properties by random or guided simulation
- several inconsistencies found
  e.g., deadline lost because of $\Delta_1 > \Delta_2$

\[
\text{now} = T_0 + \Delta_1
\]

\[
\text{output status}
\]

\[
\text{now} = T_0 + \Delta_2
\]

\[
\text{output desactivation}
\]
Validation: static analysis

- Clock reduction
  1\textsuperscript{st} version: 143 clocks reduced to 41 clocks
  2\textsuperscript{nd} version: 55 clocks, no more reduction

- Live variable analysis
  20\% of all variables are dead in each state

- Slicing
  eliminate passive processes, without outputs
## Validation: model generation

### Some results (31 processes)

<table>
<thead>
<tr>
<th></th>
<th>time deterministic</th>
<th>time non-deterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>- live reduction - partial order</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>+ live reduction - partial order</td>
<td>2201760 st. 18796871 tr.</td>
<td>n.a.</td>
</tr>
<tr>
<td>+ live reduction + partial order</td>
<td>1604 st. 1642 tr.</td>
<td>195718 st. 278263 tr.</td>
</tr>
</tbody>
</table>
Validation: model-checking

• evaluation of $\mu$-calculus formula

Property: “the stop sequence no. 3 could happen only in a flight phase”

$$\neg \mu X. <\text{EPC!Stop3}>\text{True} \lor <\text{EAP!Fire}>X$$

• construction and visualisation of bisimulation reduced models
Validation: model-checking

Property: whenever a problem is detected during the ignition of the Vulcan engine, then the whole ignition is aborted, otherwise the launcher eventually lifts off.

Graph obtained by weak bisimulation minimisation.
Outline

Key Research issues
• Modeling Real-time systems
• From application SW to implementations
• Component-based construction

The modeling framework
• Parallel composition
• Adding timing constraints
• Scheduler modeling
• Timed systems with priorities

The IF toolset
• IF notation
• Core components
• Validation
• Front ends
• Case studies

Discussion
Specific and tractable construction methodology

- Rely on a minimal set of constructs and principles e.g. combines parallel composition and restriction by priorities

- Avoid declarative formalisms such as temporal logic, LSC

- Focus on specific construction principles and rules to ensure correctness constructively, especially for safety and deadlock-freedom
Discussion: Modeling - combining behavior and priorities

Priorities prove to be a very powerful modeling tool
- they can advantageously replace static restriction
- they allow straightforward modeling of urgency and of scheduling policies
- run to completion and synchronous execution can be modeled by assigning priorities to threads
- Layered description => separation of concerns => incremental description

The IF notation is expressive enough to map compositionally most UML constructs and concepts e.g. Classes, state machines, activity groups
Discussion : validation

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