IF Tutorial

Marius BOZGA      Susanne GRAF
Iulian OBER       Laurent MOUNIER

VERIMAG
Distributed and Complex Systems Group
www-verimag.imag.fr/PEOPLE/async/IF/
Goal

Early detection of problems, concerning functional and non-functional aspects
model-based simulation, testing and verification

Context

Telecommunication protocols, Real-time embedded systems, Distributed systems, Scheduling problems,…
approach: build on the existing

User level modeling and CASE tools (SDL, UML, SCADE, …)

Translation to an intermediate language, rich enough for modeling and for validation

Optimisation and abstraction

Semantic model (state graph)

simulation
test
verification1
verification2
verification3

state explosion
motivation – language – tools – case studies - perspectives

approach: build on the existing

LOTOS

CADP

guarded commands

Optimisation and abstraction

Semantic model (state graph)

simulation

test

verification1

verification2

verification3
motivation – language – tools – case studies - perspectives

approach: build on the existing

Kronos

Timed automata

Optimisation and abstraction

Semantic model (state graph)

verification1

verification2
A good intermediate representation

• Sufficient *expressiveness*: allows to map concepts of diverse modeling languages (asynchronous, synchronous, timing,…)

• Enough *concepts*: structured representation of
  – Concepts existing in validation tools
  – Concepts exploitable for more efficient validation

• Allows *semantic fine tuning*: allows expression of alternative options of semantic variation points: time progress, execution and interaction modes,…
overview

• Motivation and challenge
• **IF: the language concepts**
  – Functional aspects
  – Non-functional aspects
• **IF: the toolset**
  – Core components
  – Model-based validation
  – Front-end tools
• Demos
• **Case studies**
• Perspectives
• **UML-based methodology for real-time systems**
  – component-based modeling
  – combination asynchronous and synchronous systems
  – relate functional and non-functional aspects

• **improve verification and test generation methods**
  – more static analysis, *abstraction* and constraint propagation
  – more compositional verification methods
  – better diagnostics facilities

• **more connections**
  – connections with *performance evaluation tools*
The IF Language

Functional Part
System description: 3 axes

IF

Processes

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

Data

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

Communications

- asynchronous channels
- shared variables
execution model

• 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 communications:
  – asynchronous signal exchanges (directly or via signalroutes)
  – shared variables

⇒ semantics can be expressed by an (infinite) LTS
**system structure**

```plaintext
const N1 = ...;  // constants
type t1 = ...;    // types

signal s2(t1, t2),  // signals

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

// processes
process P1(N0)
    ...  // data + behaviour
endprocess;

...  

process P3(N3)
    ...  // data + behaviour
endprocess;
```

**Diagram:**
- `P1(N1)`
- `P2(N2)`
- `P3(N3)`

**Annotations:**
- `s1(t1)`
- `s2(t1, t2)`
- `sr(1)`
- `signalroute`
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 clients with abort;
signalroute s1(1) #unicast #lossy
    from server to clients with grant,fail;
signalroute s2(1) #unicast
    from clients to server with req;

process server(NS) … endprocess;
process clients(NC) … endprocess;
IF processes = timed, hierarchical, finite-state automata with actions

```idl
process P1(N1);
  fpar ... ;
  // types, variables, constants, procedures
  state s0 ... ;
    ... // transition t1
  endstate;

state s1 ...;
  ... // transitions t2, t3
  endstate;

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

**Local data + local clocks**

- **Local data**
  - parameters
  - local data

- **Local clocks**
  - outgoing transitions
  - state

- **Local data + local clocks**
  - P1(N1)
state

attributes:
- #start
- #stable | #unstable

interleaving between processes can happen only on #stable states (to control transition atomicity)

the consumption of these signals is (temporarily) postponed (to control the queuing policy)
transition = deadline + optional trigger + statement list

state s0
...

deadline eager provided x!=10;
when c2 >= 4;
input update(m);
....
nextstate s1;
...
endstate;

statement = data assignment
message emission,
process or signal route creation or destruction, ...

unstable

s0 t1 s1 t2 s2

atomic!

= trigger

sequential, conditional, or iterative composition

discrete guard

timed guard

signal consumption from the process buffer
Variables:

- are **statically typed** (but *explicit conversions* allowed: \{t1\}(x))
- can be declared **public** (= shared), or not …

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

**Predefined type constructors:**

- (integer) interval: \textbf{type} fileno = \textbf{range} 3..9;
- enumeration: \textbf{type} status = \textbf{enum} open, close \textbf{endenum};
- array: \textbf{type} vector = \textbf{array}[12] of pid
- structure: \textbf{type} file = \textbf{record} f fileno; s status \textbf{endrecord};

**Abstract Data Type definition facilities …**
Signal emission:

- 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:

- `input req (f, s);`

blocking if no req signal in top of the process buffer ...
signal route = process to process communication channel with attributes, can be *dynamically* created

**attributes:**
- queuing policy: **fifo** | **multiset**
- reliability: **reliable** | **lossy**
- delivering policy: **peer** | **unicast** | **multicast**
- *delaying policy: urgent* | *delay*[\(l,u\)] | *rate*[\(l,u\)]
delivering policies

peer

unicast

multicast

to one specific instance

to a randomly chosen instance

to all instances
**example: ABP**

```
type data = range 0 .. 3;

signal get(data), put(data), ack(boolean), sdt(data, boolean);

signalroute tr(1) #unicast #lossy
   from transmitter to receiver with sdt;

signalroute rt(1) #unicast #lossy
   from receiver to transmitter with ack;

process transmitter(1) … endprocess;
process receiver(1) … endprocess;
```
system – process – communication – example – extensions

transmitter

process transmitter(1);

var t clock;
var b boolean;
var c boolean;
var m data;

state start #start ;
task b := false;
nextstate idle;
endstate;

state idle;
input put(m);
output sdt(m, b) via {tr}0;
set t := 0;
nextstate busy;
endstate;

state busy;
input ack(c);
nextstate q8;

when t = 1;
output sdt(m, b) via {tr}0;
set t := 0;
nextstate busy;
endstate;

state q8 #unstable ;
provided c = b;
task b := not b;
reset t;
nextstate idle;

provided c <> b;
nextstate busy;
endstate;
endprocess;
process receiver(1);

var b boolean;
var c boolean;
var m data;

state start #start ;
    task b := false;
    nextstate idle;
endstate;

endprocess;
dynamic creation

• remote process/signalroute creation:

```plaintext
p := fork client (true)
```

• process/signalroute destruction:

```plaintext
kill {client} (2)
kil p
```

• process termination:

```plaintext
stop
```
const NS = ..., Max = ...;
signal done(pid);
signal req();

signalroute cs(1)
  from client to server with done;

process client(0);
  fpar parent pid;
  state init #start ;
  informal "work";
  output done(self) via {cs}0 to parent;
  stop;
endstate;
endprocess;

process server(1);
  var i integer;
  var x pid;
  state idle #start ;
  provided (i < Max);
  input req();
  x := fork client(self);
  task i := (i + 1);
  nextstate idle;
  input done(x);
  task i := (i - 1);
  nextstate idle;
endstate;
endprocess;

creates a new client
receives a new request
a work is done
dies when work is finished
Several kinds of transitions …

- \( t_0 = s_0 \rightarrow s_5 \)
- \( t_1 = \text{current}_\text{state} \rightarrow s_1 \)
- \( t_4 = \text{current}_\text{state} \rightarrow \text{current}_\text{state} \) or \( t_4 = \text{current}_\text{state} \rightarrow s_5 \)

- no parallelism inside a state
- essentially a macro-notation
Use of Abstract Data Types:

```plaintext
type sqn = range 0.. N;
type sqnSet = abstract
  sqnSet Empty();
  sqnSet Insert(sqnSet, item);
  boolean isIn (sqnSet, item)
endabstract;
```

```c
typedef unsigned if_sqn_set_type;
define if_sqn_set_copy(x,y) (x)=(y)
define if_sqn_set_compare(x,y) (x)-(y)
define if_sqn_set_print(x,f) fprintf(f,"%#x",x)
define if_sqn_set_reset(x) (x)=0
if_boolean_type if_isIn_function(if_sqn_set_type p1,if_integer_type p2)
  {return (p1 & (1 << p2)) ? if_boolean_true : if_boolean_false;}
if_sqn_set_type if_Insert_function(if_sqn_set_type p1,if_integer_type p2)
  { return p1 | (1 << p2);}
if_sqn_set_type if_Empty_function()
  { return 0;}
```

**C/C++**

At the IF level only the signature is required...

... but a concrete C/C++ implementation must be provided to use the simulation tools.
C++ procedures can be used to describe data transformations:

```c++
const NUSERS = 5, NFILES = 10;

type UserIdType = range 0 .. NUSERS;

type FileIdType = range 0 .. NFILES;

type SystemStatusType = array [NFILES] of FileControlBlockType;

type FileControlBlockType = array [NUSERS] of boolean;

var updating SystemStatusType;

procedure File_Available_For_Write;

fpar in f FileIdType, in u UserIdType, in systemStatus SystemStatusType;

returns boolean;

#{ /* true if nobody is updating, maybe except u */

int uprime, result=1;
for (uprime=0; uprime<NUSERS; uprime++)

result &= (uprime==u || ! updating[f][uprime]);

return result;
#

endprocedure;
```

Checks if for all u':

updating[u'][f] implies u'<>u
The IF Language

Non-functional Part
time in system execution

- the model of time \([\text{timed automata with urgency}]\)
  - centralized → same clock speed in all processes
  - passes in stable states → transitions are instantaneous
  - depends on the system state → precisely timed behavior
• real-valued clocks
  – operations: set, reset (deactivate)
• timed guards
  – comparison of a clock to an integer
  – comparison of a difference of two clocks to an integer

SPIN workshop
April 2, 2004

state send;
  output sdt(self,m,b) to {receiver}0;

  nextstate wait_ack;
endstate;

state wait_ack;
  input ack(sender,c);
  ...;
...;
endstate;
linking time and system progress

- 3 types of urgency for time-guarded transitions
  - **eager** transitions: urgent as soon as they are enabled
    block time progress
  - **lazy** transitions: never urgent
    always allow time progress
  - **delayable** transitions: urgent when about to be disabled by time progress
    allow time progress otherwise

```plaintext
process channel;
state get;
input SDT(p,q,r);
nextstate forward;
endstate;

state idle;
input PUT(p) // from ENV;
nextstate forward;
endstate;

state wait_ack;
when t_repeat = 0;
input SDT(p,q,r);
nextstate get;
endstate;
```
semantics of urgency

x: clock

1<x<3 urgency

x:=2

urgency

eager

delayable

lazy
resources

- mutually exclusive access to a physical or logical resource by concurrent IF processes
  - acquisition: precondition to a transition – models passive wait
  - release: action – executed when resource is not needed any more
dynamic priorities

- partial priority order between processes based on global state

  \[ \text{priority}\_\text{rule} : p_1 < p_2 \text{ if condition}(p_1,p_2) \]

- \( p_1 \), \( p_2 \) are free variables ranging over the active process set
- semantics:

  \textit{among enabled processes, only maximal elements execute}

- applications: scheduling policies
  - fixed priority:
  - run-to-completion:
  - EDF:
  ...

IF Toolset
architecture

IF Tutorial

SPIN workshop
April 2, 2004
IF Toolset

Core Components
- language API
- exploration API
- simulator design
language API – exploration API – simulator design

language API

• gives programming access to the AST of an IF specification

• AST represented as a collection of C++ objects
language API – exploration API – simulator design

AST overview

Diagram showing the structure of SystemEntity: Signal -> Signalroute -> ProcessEntity -> Procedure -> Type
- Variable
- State
- Constraint
- Expression
- Constraint
- InputAction
- BlockStatement
- Action
- Expression

Overall note: For IF Tutorial, SPIN workshop, April 2, 2004
# include "model.h"

void main() {
  IfObject::Initialize();
  // parse the input
  IfSystemEntity* sys = Load(stdin);
  if (sys != NULL)
    sys->Compile();
  // for each process...
  for(int i = 0; i < sys->GetProcesses()->GetCount(); i++) {
    IfProcessEntity* proc = sys->GetProcesses()->GetAt(i);
    printf("%s:", proc->GetName());
    // for each local variable...
    for(int j = 0; j < proc->GetVariables()->GetCount(); j++) {
      IfVariable* var = proc->GetVariables()->GetAt(j);
      // find if the variable is used in some state
      int used = 0;
      for(int k = 0; k < proc->GetStates()->GetCount(); k++) {
        IfState* state = proc->GetStates()->GetAt(k);
        used |= state->Use(var);
      }
      if (! used)
        printf("%s ", var->GetName());
    }
  }
}
applications

- static analysis
  - live variables, slicing, dead code
- code generation
  - simulation code, application code
- translation
  - if2pml (by Eindhoven TU)
- pretty printing
  - if2if, if2dot, if2html
exploration API

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

- 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
states are global (system) configurations

- gray-box structural representation as set of local (process) configurations (instances)
- the content of each process configurations can be accessed
  - process identifier (pid)
  - control state pointer
  - queue of pending input signals
  - local variables and parameters

labels record observable events occurring on transitions

- structural representation as a list of events
- each event can be accessed
  - issuing process
  - event type (INPUT, OUTPUT, FORK, etc.)
  - type dependent auxiliary information
language API – exploration API – simulator design

LTS traversal

- **Driver**
  - void explore(Config* p, Label* a, Config* q) = 0;

- **Engine**
  - Config* start();
  - void run(Config* p);

**Diagram:**
- `engine.run(p)`
- `driver.explore(p, a_1, q_1)`
- `driver.explore(p, a_2, q_2)`
- `driver.explore(p, a_n, q_n)`
- Nodes: `p`, `a_1`, `a_2`, `a_n`, `q_1`, `q_2`, `q_n`

**Text:**
- Interface class: the user must implement the `explore` method to handle successors.
- Concrete class: providing traversal functions.
language API – exploration API – simulator design

an example: bfs search

1. #include "simulator.h"
2.
3. class BfsExplorer : public IfDriver {
4.    static const int REACHED = 1; // reachable state marking
5.    Queue m_queue; // the queue of unexplored states
6.
7. public:
8.    // successor handler: append target state to the queue, if not yet reached
9.    void explore(IfConfig* source, IfLabel* label, IfConfig* target) {
10.        if (! (target->getMark() & REACHED) )
11.            { target->setMark(REACHED); m_queue.put(target); }
12.    }
13.    // visit one state i.e, print it on the screen
14.    void visit(IfConfig* state) {
15.        state->print(stdout);
16.    }
17.    // visit all states, main bfs loop
18.    void visitAll() {
19.        IfConfig* start = m_engine->start();
20.        start->setMark(REACHED); m_queue.put(start);
21.        while (! m_queue.isEmpty()) {
22.            IfConfig* state = m_queue.get();
23.            visit(state);
24.            m_engine->run(state);
25.        }
26.    }
27.};
• Debugging
  – interactive, random simulation

• Model-checking
  – exhaustive model generation
  – on-the-fly $\mu$-calculus evaluation
  – model exploration with observers

• Testing
  – test case generation
  – on-the-fly timed testing

• Optimization
  – shortest path computation
simulator design

- goal: offer primitives to explore the state space of IF specifications in an exhaustive manner

- main functionalities
  - simulate the process execution
    - inter-process communication
    - process creation / destruction
    - control of simulation time
  - handle non-determinism
    - asynchronous execution
    - internal non-deterministic choices
    - open environment
  - state space representation
language API – exploration API – simulator design

architecture

IF specifications

compiler

executable instances
behavior and representation

open exploration platform

application specific

predefined
(time, channels, etc.)

asynchronous execution

dynamic scheduling

state space representation
1\textsuperscript{st} layer: emulate asynchronous parallel execution

- ask in turn each instance to execute its enabled transitions
  - ensures atomicity at level of instance transitions
- when an instance is executing provides
  - message delivery, shared variable update
  - global time constraints check and clocks update
  - dynamic instance creation and destruction
  - record generated observable events
- get informed when a local step is finished and
  - take a snapshot of the global configuration and store it
  - send the successor to the 2\textsuperscript{nd} layer (dynamic scheduler)

obtain global (system) steps from local (process) steps
2\textsuperscript{nd} layer: dynamic scheduling

- collect all potential global successors
- filter them accordingly to dynamic priorities
  - evaluate each priority constraint
  - if applicable on current state
    remove successors produced by the low priority instance
- deliver the remaining set to the user application through the exploration API
**Language API** – **Exploration API** – **Simulator Design**

**Execution Control**

Active Instances:
- I₁:P₁
- I₂:P₁
- I₁:P₂
- I₂:P₂
- Iₖ:Pᵢ
- I₁:Time

Execution Control:
- Asynchronous execution
- Dynamic scheduling

Control Actions:
- Create
- Run
- Set, reset
- Step
- Explore
simulation time

at simulation, time is a dedicated process instance handling
- dynamic clock allocation (set, reset)
- represent clock valuations
- check time constraints (timed guards)
- compute time progress conditions w.r.t. actual deadlines and
- fire time transitions, if enabled

two concrete implementations are available (other can be easily added)

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

ii) dbm time
- clock valuations represented using varying size difference bound matrices (DBMs)
- time elapse is symbolic
- non-convex time zones may arise because of deadlines: they are represented implicitly as unions of DBMs
state representation

- configurations
- configuration chunks
- instances
- queue contents
- messages

state storage is completely done by the simulator

structural representation of configurations offering maximal sharing

unique tables implemented as hash tables with collision or search trees (splay trees or 2-3 trees)
<table>
<thead>
<tr>
<th>Open/Cæsar</th>
<th>exploration API i.e, labeled transition system interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>System C</td>
<td>simulator architecture i.e, open platform + running objects</td>
</tr>
<tr>
<td>Kronos, Uppaal</td>
<td>symbolic time representation and operations using DBMs</td>
</tr>
<tr>
<td>BDDs</td>
<td>state space representation</td>
</tr>
</tbody>
</table>
IF Toolset

Model-Based Validation
- model checking
- test generation
- optimization
- static analysis
specify system properties in an operational way

observes
- events
- system state
- time

states
- normal / error / success

properties
- linear, timed
- safety/liveness

semantics
- weakly synchronized composition (i.e. greater priority than the system)
observation and actions

- **state** observation
  - variables, queues, process-in-state

- **event** observation
  - event types: INPUT, OUTPUT, FORK, KILL, DELIVER, …
  - retrieve data related to event
    - signal parameters
    - created process’ pid…

- **actions**
  - internal: local variables, etc.
  - control system simulation/exploration
    - cut the exploration
    - inject signals, mutate variables

Verification: reachability (safety)
**μ-calculus evaluation**

- **alternating-free fragment**
  \[ \phi ::= T | X | <a>\phi | \neg\phi | \phi \land \phi | \mu X. \phi(X) \]
  where \( a \) denotes a regular expression on labels

- **macros** available to describe complex formula e.g,
  \[
  \text{all } \phi \equiv \nu X. \phi \land [*]X \\
  \text{pot } \phi \equiv \mu X. \phi \lor <*>X \\
  \text{inev } \phi \equiv \mu X. \phi \lor <*>T \land [*]X
  \]

- **IF toolset** includes an 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”)

behavioral relations

• LTS comparison:
  – equivalence relations (“behavior equality”):
    System ≈ Specification
  – preorder relations (“behavior inclusion”):
    System ≤ Specification

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

• several relations available:
  weak/strong bisimulation, branching, safety, trace equivalence

• use of CADP as back-end:
  aldebaran, bcg_min
model checking – test generation – optimization – static analysis

example

reduction w.r.t. branching bisimulation
Conformance testing for distributed applications

Two implementations:
- TGV (Irisa/Verimag) for Lotos, SDL, UML and IF
- TestComposer (Telelogic), inside ObjectGeode
model checking – test generation – optimization – static analysis

principle of TGV

• System architecture:

• Specification (IF,…)

• Test purpose: property

⇒ TGV computes test cases:

A, C: controllable
B, D, Y: observable
W, X, Z: internal

Exhaustive system behaviour
(in terms of A,B,C,D,W,X,Y,Z)

IF Tutorial

SPIN workshop
April 2, 2004
test case generation in TGV

 elim. non observables, determinize, eliminate conflicts

TP || S

elim. non observables, determinize, eliminate conflicts

TC

model checking – test generation – optimization – static analysis
TGV results

• advantages of automatic test case generation:
  – less error prone
  – less time consuming
  – applicable to real systems

• problems of automatic test case generation:
  – manual tests are symbolic -> less test cases
  – detailed formal specification is needed

• AGEDIS IST project (integration of IF/TGV inside a complete testing framework):
  – model specification in UML, translation to IF
  – test generation with TGV
  – test execution on Java programs with Spider (IBM)
• there are (user defined) costs associated to transitions of the semantic model of IF specifications e.g., waiting times

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

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

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

• philosophy
  – 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
live variables

a variable is dead in 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 communication primitives
asynchronous communication via queues
parameter passing at process creation
live variables are propagated both intra and inter processes!

exploit live variables
transform IF specification 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

y := z+2
reset y
reset y

y not used here

y := 3*x

?m(x, y)
reset y
reset y
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 specifications by
- removing processes never created
- removing signals never sent
- removing unreachable control states and control transitions

the gains are
- reduce the size of the specification
- enable more reduction by live analysis
- strongly preserve the initial behavior, under the given assumptions

process P(1)
- ?a
- ?b
- fork Q
- !b
- !c

process Q(0)
- ?b
- fork R

process R(0)
- ?c
- !a

provides only “a” signals to the process P

model checking – test generation – optimization – static analysis
variable elimination

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 specifications by
- removing assignments to undefined variables
- removing undefined signal and process parameters
- relaxing guards involving undefined variables
obtain a conservative abstraction of the initial specification i.e., including all the behaviors of the initial one

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

i

[j=N][i<N]
i:=0

[i<N]
i:=i+1

b(i)

k, x

[k odd][k even]

x:=x+k
x:=0

?b(k)

i

b

x

reset x
x:=0

?b
IF Toolset

Front-Ends
- sdl2if
- uml2if
Specification and Description Language

- formal specification language for distributed systems
  - concurrent processes (Extended FSM)
  - asynchronous buffered communication

- widely accepted in telecommunication area
  - ITU standard, revised every 4 years (‘88 – ’00)
  - development methodologies
  - commercial tool support
SDL concepts

• hierarchical structuring mechanism
  – system, blocks, processes, services (agents)
• high level process description language
  – nested states, structured transitions
  – various elementary triggers and actions
  – procedures
• dynamical features
  – process creation and destruction
• timing aspects
  – timer concept, global time (now)
• object-oriented features
  – parameterization, inheritance

• formal semantics defined in terms of Abstract State Machines (ASM)
SDL translation

• translation of SDL into IF is straightforward
  – direct mapping of SDL elements into IF ones
  – at origin, IF was an intermediate representation for SDL

• but there exists some limitations
  – hierarchical system decomposition
  – procedures and procedures calls
  – complex data types
  – arbitrary use of now in expressions
sdl2if relies on a full SDL parser provided by Telelogic AB.

Several transformations are applied on the SDL/AST prior to its translation (i.e., SDL’xx reduced to SDL’88).
OMG’s standard modeling language

- developed since 1998, current versions: 1.4, 2.0
- widely accepted in industry, wide tool support
- complex (10 types of diagrams, ≈150 types of concepts)
  - mixes declarative / imperative, OO, synchronous/asynchronous, aspect oriented, …
  - for requirements / design
- informal semantics

Class diagrams

State diagrams

Sequence diagrams
our focus: real-time and embedded systems (OMEGA)

• cover operational specifications
  – classes with operations, attributes, associations, generalization, statecharts; basic data types
• define a particular execution model
  – a notion of active class
  – active objects define activity groups
  – run-to-completion, group stability
• communication and behavior
  – primitive operations – procedural, stacked
  – triggered operations – embedded in state machine, queued
  – asynchronous signals
• define an Action Language
a mapping of OO concepts to (extended) automata

• 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
tool architecture

Rhapsody
Rose
Argo
Objecteering

XMI 1.0/1.1 (UML 1.4 + stereotypes)

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

IF 2.0 TOOLBOX
IF spec

IF Tutorial
simulation / verification interface

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

telecommunication protocols
embedded and distributed software
manufacturing problems
asynchronous circuits
protocols – embedded – distributed – manufacturing - circuits

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
pragmatic general multicast

**Protocol specification**

**Key features**
- Real-time data transmission for multimedia multicast using tree architecture
- Generalised sliding window for error recovery
- Negative acknowledgment
- Important timing constraints
- Many parameters (buffer lengths, delays)

**SDL specification (~3500 lines)**
- Formalize the IETF draft
- Developed by France Telecom
- Translated completely using sdl2if

**Protocol requirement**
- Any receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data loss
model checking

**initial model**
limited by the size of state space i.e.,

the configuration with 1 sender, 1 network element, 2 receivers, 2 messages sent, arbitrary loss, has more than 200000 states, 800000 transitions

**abstract model**
abstract the multicast tree as a linear structure + noise processes

scenarios with up to 12 messages sent and arbitrary losses have been considered

safety properties have been verified on the fully generated state space

an error detected w.r.t. to the transmission and recovery of the last packet in a sequence

---

**model exchange**

PGM models developed in IF have been exchanged among ADVANCE partners

many other techniques applied on PGM: symbolic reachability, regular model checking, parameter synthesis
embedded software

Ariane 5 Flight Program
joint work with EADS Launchers

K9 Rover Executive
Joint work initiated by EADS-LV to evaluate verification techniques and tools through a specific case-study.

**flight program specification**

built by reverse engineering by EADS

high level, non-deterministic, abstracts the whole program as communicating extended finite-state machines

~3500 lines of SDL code

**flight program requirements**

**general requirements**

– no deadlock, no timelock
– no implicit signal consumption

**overall system requirements**

– flight phase order
– stop sequence order

**local component requirements**

– activation signals arrive eventually in some predefined time intervals
### ariane 5 flight program

#### translation
- the SDL specification has been translated completely into IF using sdl2if
- urgency of transitions has been explicitly defined to achieve the intended behavior

#### model generation
- partial order reduction needed
- 31 interleaved processes
- the full state space can be constructed
- 195718 states, 278263 transitions

#### model exploration
- random or guided simulation
- several inconsistencies found

#### static analysis
- clock reduction
  - 1<sup>st</sup> version: 143 clocks reduced to 41 clocks
  - 2<sup>nd</sup> version: 55 clocks, no more reduction
- live variable analysis
  - 20% of all variables are dead in each state
- slicing
  - eliminate passive processes, without outputs
- evaluation of µ-calculus formula
  - property: the stop sequence no. 3 could happen only in a flight phase
  - \[ \rightarrow \mu X. \langle EPC!\text{Stop3}\rangle \text{True} \land \langle EAP!\text{Fire}\rangle \text{X} \]
- build bisimulation reduced models
  - 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
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
mq series integration broker

specification
lightweight publish/subscribe protocol for integrating devices with WebSphere Integration Broker™

modeling
the protocol has been modeled using the AGEDIS Modeling Language AML – an UML profile for testing
IF is an intermediate representation for AML

test generation
several tests have been extracted successfully using TGV / AGEDIS
test directives combines functional goals (e.g., connection establishment, publishing, notifications) and coverage criteria (e.g., return values for methods)

test execution
generated tests have been applied on concrete implementations using SPIDER, the AGEDIS Test Execution Engine
(injected) errors have been discovered
Job-shop Scheduling

Axxom Lacquer Production

case study proposed in AMETIST IST-2001-35304
chemical industry problem

There are 29 lacquers to be produced, each one in some predefined time interval [earliest-start date, due date].

Lacquers are of 3 different types, each type has a specific production flow, characterized by the resources involved, processing times, flow constraints, etc.

Problem: find an optimal schedule i.e., with minimal delays for the production of 29 lacquers.

IF-based solution

Reduce the scheduling problem to a minimal path cost extraction problem:

- Model each lacquer as an IF process encoding resource allocation/deallocation order, basic task duration, additional flow constraints.
- Model the production plan as the parallel composition of lacquers automata + resources.

The optimal schedule correspond to the minimal cost path leading from the initial state to a state where all lacquers have completed successfully.

lacquer type

<table>
<thead>
<tr>
<th>Mix</th>
<th>Lab</th>
<th>Dk</th>
<th>Tp2</th>
<th>Hdl</th>
<th>Dk</th>
<th>Lab</th>
<th>Abf</th>
</tr>
</thead>
<tbody>
<tr>
<td>4h</td>
<td></td>
<td>6h</td>
<td>4h</td>
<td>4h</td>
<td>6h</td>
<td>4h</td>
<td></td>
</tr>
<tr>
<td>6h</td>
<td>14h</td>
<td>12h</td>
<td>61h55</td>
<td>75h58</td>
<td>8h</td>
<td>40h</td>
<td>35h11</td>
</tr>
<tr>
<td></td>
<td>1h</td>
<td>3h</td>
<td>1h</td>
<td>1h</td>
<td>1h</td>
<td>1h</td>
<td>1h</td>
</tr>
</tbody>
</table>
finding an optimal path

the search space is huge because of
the interleaving of 29 processes using
more than 73 clocks!

several heuristics have been applied
at source level to reduce the search:
  avoid lazy runs i.e, remove useless waiting
  from schedules
  avoid phase overtaking between jobs
  (lacquers) of the same type i.e, ensure a
  pipelined execution
  enforce minimal separation time between
  jobs of the same type

It take 15” to find that an optimal 0-
delay schedule exists on the model an
to extract it using the IF optimizer

IF outperform standard MILP (Mixed
Integer Linear) approaches on the same
case study

but still not all the difficulties of the real
case study have been considered e.g,
  batch splitting, operating hours, sequence
  depending costs, performance factors
asynchronous circuits

timing analysis


functional validation

asynchronous circuit problem
knowing individual gate latencies, find the maximal stabilization time of the circuit, for an arbitrary change of inputs

IF-based solution
model each gate as a timed automaton and the circuit as the product of gates
the maximal stabilization time correspond to the maximal delay path leading from the initial state to some next stable state
this method is exact, and therefore more accurate than usual methods which ignore the data part (no false paths !)
nevertheless, we are limited by the size of the circuit (number of gates)