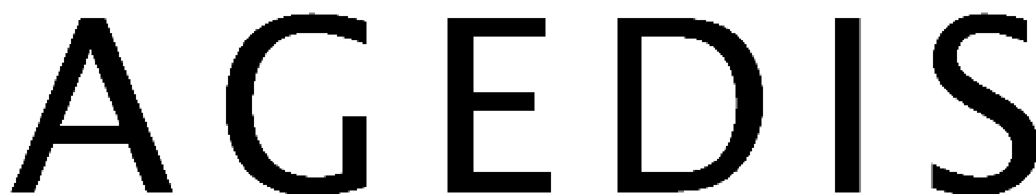
Intermediate Format 2.0 Language with Test Directives

Deliverable 3.2

Owner	Sergey Olvovsky
Approvers	Laurent Mounier
Status	Approved
Date	10/02/2002



Change History

Date	Author	Summary of chanes
10/02/2002	Marius Bozga, Sergey Olvovsky	IFE with test directives 1st release
23/10/2002	Sergey Olvovsky	Document format changed, content unchanged

Contents

1 Introduction....4
2 Extended Automata....4
3 Design Decisions....5
4 Examples....6
4.1 Parallel Server....6
4.2 File System....8
4.3 Token Ring Protocol....10
5 Discussion....13
6 Test Generation and Language Mapping Extension to IF....14
6.1 Test Directives....14
6.2 Test Architecture....17
6.3 Mapping Information....17
References....18
A Structure....19
A.1 System....19
A.2 Process....19
A.3 Signalroute....20
A.4 Signal....20
B Behavior....20
B.1 State....20
B.2 Transition....21
B.3 Action....22
C Data....22
C.1 Constant....23
C.2 Type....23
C.3 Variable....23
C.4 Expression....24
C.5 Constraint....24
D External....25
D.1 Procedure....25
E Test Directives....25
F Test Architecture....27
G Mapping Information....28

1 Introduction

The Intermediate Format 2.0 and Test Directives Language (IF2.0+TD) is being developed by the AGEDIS project consortium. AGEDIS stands for Automated Generation and Execution of Test Suites for Distributed Component-based Software.

This proposal builds upon the definitions of the Intermediate Format (IF) language and the Common Description Language by Verimag. This proposal expands the descriptive part of the IF/CDL languages and includes the new test generation related features.

The Common Description Language is an extension of the IF language developed at Verimag in order to model asychronous communicating real-time systems. Since its definition in 1998, the IF language has been continously improved and has become the ***interchange format*** between a set of validation tools dedicated to real-time systems i.e, the so-called IF validation environment. In this project we introduce extensions to the IF language, that make it suitable to model a much wider range of systems, including distributed software systems. In particular, we extend the IF language, and more generally, the IF environment to be able to model and analyze systems that rely upon dynamic process creation and dynamic data types. In addition we provide a separate test directives notation, nescessary to support model based test generation.

The document is organized as follows. Section **2** presents the extended automata model i.e, the formal basis of the IF2.0 language. In section **3** we summarize the main design decisions. Section **4** gives an incremental presentation of the language through several examples. In section **5** we discuss the missing features and the status of the tools. Finally, Section **6** gives the description and the language syntax of the test directives extension. The annexes **A**, **B**, **C** and **D** give the complete language syntax in BNF and some more syntax-oriented explanations.

2 Extended Automata

The formal basis for our language is a dynamic version of extended automata.

We focus on systems composed from several components, hereafter called processes, running in parallel and interacting through point-to-point message-passing. The number of processes may change over time: they may be created and destroyed dynamically using specific actions, during the system execution.

Each process is an extended timed automaton. It has a unique process identifier (***pid***) value and local memory consisting of variables (including clocks), control state and a queue of pending messages (received and not yet consummed). Processes move from one control state to another by executing transitions. In a control state, transitions are trigerred by the presence of some message in the input queue and/or some (possibly timed) condition on variables. In addition, transition bodies are ***sequential*** programs

consisting of elementary actions such as variable assignments, clock settings, message sending, process creation/destruction, etc.

The semantics of the extended automata model is given by the graph of its executions. This graph is obtained by the ***interleaved*** execution of processes, that is, one process is executing a transition at a time, while the others are waiting. In other words, we consider process transitions to define ***atomic***, non-interruptable, execution steps.

The semantics of time is as for timed automata: time may progress only in states (i.e, all running processes wait in some state before selecting and executing some transition) and transitions take zero time to be executed. In order to control the progress of time, or equivalently, the waiting time in states, we rely on transition urgencies - explicit deadlines attached to transitions defining when these transitions ***must*** be executed.

3 Design Decisions

The IF2.0+TD language inherits from IF some of its main features, in particular:

- ***structuring concepts:*** systems consist of processes, running in parallel and communicating through message passing via communication buffers;
- ***communication primitives:*** more exactly, processes may communicate either through signal exchange (directly or via signalroutes) or through shared variables;
- ***real-time primitives:*** each process may use several clocks to measure time during the execution and, in addition, transitions may be guarded with time constraints (depending on clocks) and decorated with explicit (***eager, delayable, lazy***) deadlines;
- ***open systems:*** the language includes the concept of an open communication channel, connected to the environment, and transporting messages between it and the system;
- ***non-determinism:*** processes may be non-deterministic i.e, more than one transition may be enabled at some control state, and all situations have to be considered at execution;
- ***complex data types:*** the language provide several ***type constructors*** such as enumeration, range, array, record, abstract as well as ***predefined basic types*** in order to simplify complex data description and manipulation;

In order to fit the new requirements, the IF2.0+TD language improves the IF language in several directions, the most important being listed below:

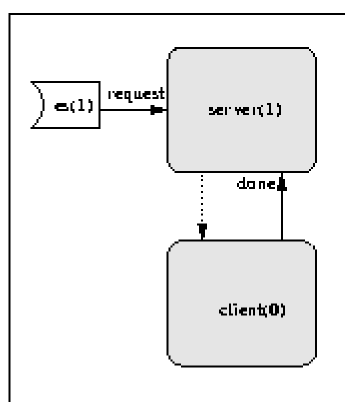
- ***parameterization:*** using integer constants it is now possible to parameterize data types (i.e, size of arrays), system configurations (i.e, number of instances), timing behavior (i.e, clock constraints)...;
- ***dynamic features:*** the language include dynamic creation and destruction of process and signalroute (channel) instances. This makes the system configuration ***dynamic***, that is, the number of components running (and in turn, the number of clocks ...) may change during execution;

- ***structured control:*** the IF2.0 language integrates ***hierachical states*** (to structure automata), including sequential and parallel states and entry/exit actions in states (UML style).
- ***composed transitions:*** basic control statements such as ***if-then-else*** and ***while-do*** are provided to structure automata transitions;
- ***external code integration:*** the IF2.0 language provides a simple and elegant way to abstract complex transformations on data through the integration of external code within ***procedures***. The external code to be provided depends on the tools used i.e, an executable implementation in order to simulate and model check, or a first-order axiomatic definition in order to use it inside a prover, etc.
- ***transition priority:*** transitions, leaving a state, may have priority information, which, if specified, will define the enabled transition to be taken.
- ***variable visibility:*** a public variable modifier makes an internal process variable accessable from outside the process.

4 Examples

4.1 Parallel Server

This example describes a parallel server which can handle at maximum N requests simultaneously. Thus, when possible, for an incoming `request` message (received from the environment) a client process is created. The server keeps the number of running clients in a local variable. Client processes are quite simple: once created, they work for a while, and when finished they send a `done` message back to the server. The architecture of this application and the corresponding specification are presented in figure 1.



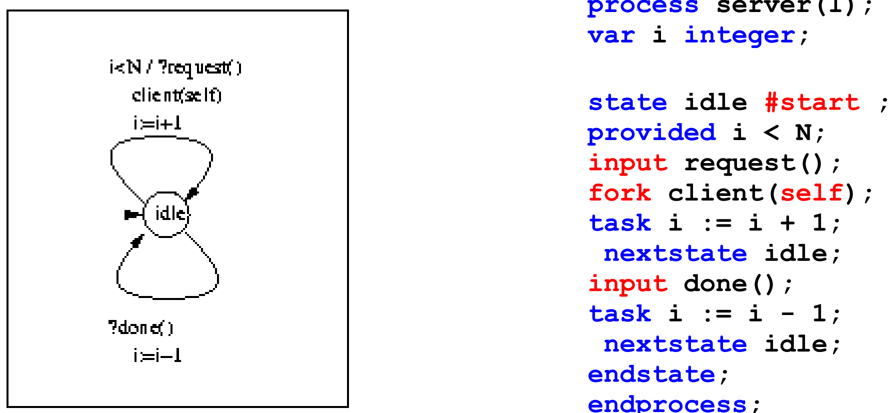

```
system server;
const N = 2;

signal request();
signal done();

signalroute es(1)
from env to server
with request;

process server(1);
...
endprocess;
process client(0);
...
endprocess;
endsystem;
```

Figure 1: Architecture specification of parallel server application.

Basically, the specification contains several definitions. First of all, the constant **N**, which is a parameter for this specification, gives the maximum allowed number of

clients running in parallel. Then, the signals **request** and **done**, used to communicate respectively between the server and the environment, and between the clients and the server, are defined. The signalroute **es** is an ***open*** (**from env**) communication channel transporting the requests. Finally, there are descriptions of respectively the **server** and **client** processes. The numbers after process and signalroute declarations indicate how many instances exist in the inititial system configuration.

Let us focus now on the **server** process definition. It has one local integer variable **i**, counting the number of running clients. Its behavior is described by a one-state automaton with two transitions. The first transition models the reaction at the incoming requests: if the counter does not exceed the maximal allowed value, a new client is created and the counter is increased. The second transition models the reaction at termination signals by decreasing the counter. The automaton and the corresponding specification are both presented in figure 2.

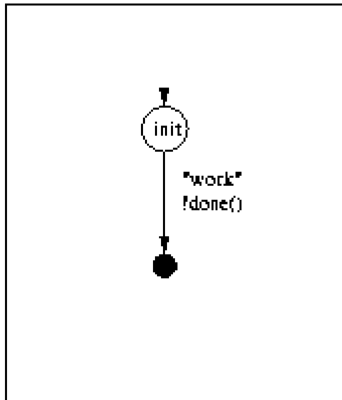



```
process server(1);
var i integer;

state idle #start ;
provided i < N;
input request();
fork client(self);
task i := i + 1;
 nextstate idle;
input done();
task i := i - 1;
 nextstate idle;
endstate;
endprocess;
```

Figure 2: Behavior specification of **server** processes.

In the server specification, the ***#start*** construct is an ***option*** string which indicates that the **idle** state is the initial state of the automaton. Let us remark on the use of the ***self*** construct, allowing one to obtain the process identifier of the **server**. Here, this value is then given as parameter to each new client instance created (through the fork instruction).

Finally, figure 3 gives the specification of **client** processes. Once created, they perform some informal **work** then send back to the parent the **done** signal. We recall here that the communication is asynchronous point-to-point. This means that, on one hand, in order to send some message, one has to know apriori the identity of the receiver process (hence, the need for the **parent** parameter which stores the identity of the server, who creates the client). On the other hand, asynchronous means that the sender is never blocked by the availability of the receiver, the message being imediately delivered to the receiver (i.e, stored in its input-queue of pending messages). Finally, the stop construct denotes the (auto) destruction of the process instance.

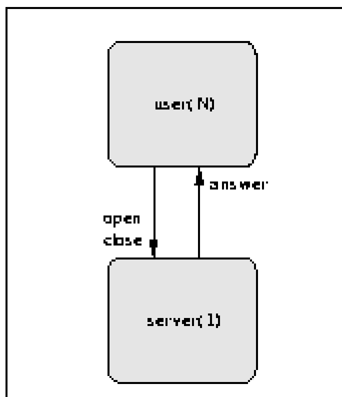



```
state init #start ;
informal "work";
output done()
to parent;
stop;
endstate;
endprocess;
```

Figure 3: Behavior specification of `client` processes.

4.2 File System

The second example describes a simple filesystem with several users communicating through a file server. The users address read/write requests to the server and wait for the corresponding accept/reject answers. The server policy is parameterized by two external procedures ***FileAvailableForRead*** and ***FileAvailableForWrite***, described here by external C code. The global architecture is presented in figure 4.



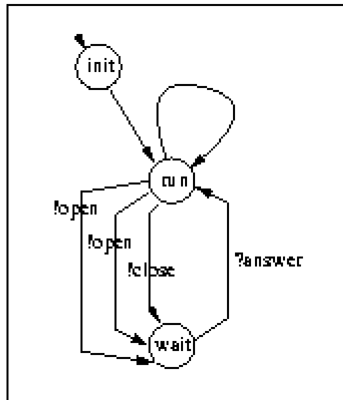

```
system filesystem;
const N = 3;
```

```
const M = 4;
type User = range 1 .. N;
type File = range 1 .. M;
type Mode =
 enum Read, Write endenum;
type = Result
 enum Accept, Reject endenum;

signal open(File, Mode, User);
signal close(File, User);
signal answer(Result);

process user(N);
 ...
endprocess;
process server(1);
 ...
endprocess;
endsystem;
```

Figure 4: Architecture specification of the file system application

The specification is parameterized by two values: `N`, the number of users and `M`, the number of files in the filesystem. It contains the definition of several application specific data types, respectively, integer domains (`User` and `File`), and the enumerated lists (`Mode` and `Result`). The specification provides description of the communication signals `open`, `close` and `answer`. Finally, there is the description of the processes in the system, respectively `user` and `server`. At this point, let us notice the number of `user` process instances is defined to be `N`.

Figure 5 describes the `user` processes. Basically, after an initialization phase they send non-deterministically `open` and `close` requests to the file server.




```
process user(N);
var u User;
var s pid;
var f File;
var r Result;
state init #start;
task u := {integer} self + 1;
task s := {server} 0;
task f := 1;
 nextstate run;
endstate;
state run;
output open(f, Read, u) to s;
 nextstate wait;
output open(f, Write, u) to s;
 nextstate wait;
output close(f, u) to s;
 nextstate wait;
task f := f % M + 1;
 nextstate -;
endstate;
state wait;
input answer(r);
nextstate run;
endstate;
endprocess;
```

Figure 5: Behavior specification of the user processes.

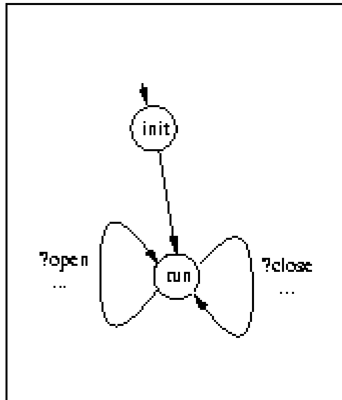
Let us focus on the initialization transition. The first two assignments illustrate the conversion operators {-} allowing to convert (in this example) integer values to process identifiers and back.

In order to understand these operators, imagine that the process identifier values are couples of the form [process,integer]. For instance [user,7] is a valid process identifier for an `user` process instance. Now, the operator {integer} applied to a pid value will extract the integer component of it (for the example above, {integer} [user,7] = 7). Conversely, the operator {process} applied to an integer will construct a pid value (example, {user} 7 = [user,7]).

In the user example, the expression `{integer} self` extract the integer value corresponding to the `self` process identifier, hence, a number between 0 and `N`-1, depending on the instance which execute the transition. Conversely, the construction `{server} 0` means the process identifier of the server instance.

Finally, remark also the construction `nextstate -;` which means remaining in the same state, where the transition begins.

The server process is sketched in the figure 6. Briefly, it stores the file system status in a two-dimensional array, which is modified accordingly opening and closing requests addressed by the users.



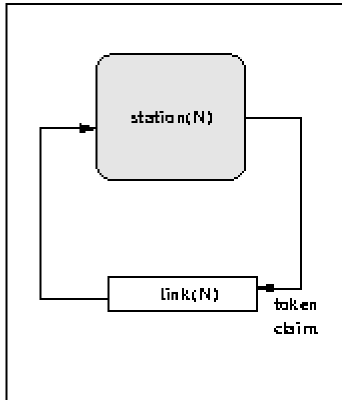

```
type FileStatus = enum
 Closed, Reading, Writing
endenum;
type FileControl =
 array [N+1] of FileStatus;
type FileSystem =
 array [M+1] of FileControl;

var fs FileSystem;
...
procedure FileAvailableForWrite
fpar in f File,
 in u User,
 in fs FileSystem;
returns boolean;
{#
 /* external C code */
 return ...;
#}
endprocedure;
...
state init #start ;
task f := 1;
while f <= M do
task u := 1;
while u <= N do
task fs[f][u] := Closed;
task u := u + 1;
endwhile
task f := f + 1;
endwhile
nextstate run;
endstate;

state run;
...
input open(f, m, u);
...
call
FileAvailableForWrite(f,u,fs);
...
input close(f, u);
 ...
endstate;
```

Figure 6: Behavior specification of the file server processes.

Let us notice the local definition of the **`FileSystem`** type, which is a two-dimensional array depending on both parameters **`N`** and **`M`**. Also, remark the definition of the external procedure ***FileAvailableForWrit***e, which implements the (possible complex) condition under that files are available for writing (e.g, ***true if nobody is writing, except maybe the same user*** **`u`**). Finally, let us observe the initialization transition. It gives an example of using the **`while`** statement, here, in order to initialize the file system status.

4.3 Token Ring Protocol

The last example we consider is the leader election protocol in distributed ring networks. The architecture of the system is presented in figure 7.




```
system token;

const N = 2;
const D = 4;

signal token();
signal claim(integer, boolean);

signalroute link(N)
#lossy #delay[1,2]
from station to station
with token, claim;

process station(N);
...
endprocess;

endsystem;
```

Figure 7: Architecture specification of Token Ring Protocol.

The specification is parameterized with two values, respectively **`N`**, the number of stations in the ring and **`D`**, the average transit time for a message to make a complete tour of the ring. Two signal definitions are provided, respectively for **`token`** and **`claim`** signals. The later signal transports two values, an integer and a boolean one. The specification includes the definition of **`link`** signalroutes, which model the communication paths between stations, and the definition of **`station`** processes.

For clarity, one can view (internal) signalroutes as specialised processes for the delivery of messages between ***normal*** processes. That is, messages transit through signalroute instances from one process instance to another process instances(s). The behavior of signalroutes does not need to be explicitly defined by the user, it is implicitly defined by the following options:

queueing policy

> denotes how the messages in transit are tackled by the signalroute. Two policies are available respectively, **`#fifo`**, order-preserving, uses a queue-based storage and **`#multiset`**, no order-preserving, uses a multiset storage.

delivery policy

> denotes how the messages are delivered. Three options are available here, respectively, **`#peer`**, which means delivery to the instance indicated in the output action using the **`to`** construct, **`#unicast`**, means delivery to one of the running instances and **`#multicast`**, means delivery to all running instances existing at the signalroute endpoint.

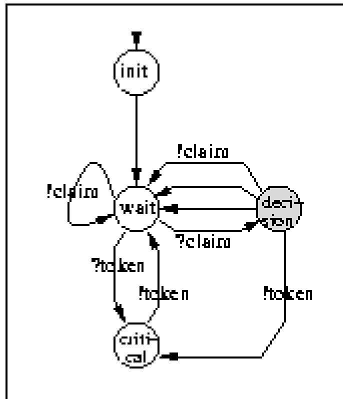
reliability

> could be either **`#reliable`**, which means that messages are not lost, or **`#lossy`**, which means that messages could be lost when transiting the signalroute.

delay policy

> denotes the delay associated with the signalroute and can be respectively, **`#urgent`**, means immediate (0-time) delivery, **`#delay[a,b]`**, means that any message entering the signalroute will eventually leave it after **`a`** and not later than **`b`** units of time, and **`#rate[a,b]`**, means that it takes between **`a`** and **`b`** units of time per message to be delivered by the signalroute.

In the token ring example, we considered that we have **`N`** signalroutes between stations, that all of them loose messages (#lossy) and delay messages between 1 and 2 units of time.




```
process station(N);

var worried clock;
var round boolean;
var i integer;

...

state init #start ;
task i := {integer}self;
task round := true;
set worried := 0;
nextstate wait;
endstate;
state wait;
input token();
reset worried;
nextstate critical;
when worried>=D;
set worried := 0;
output claim(i, round)
via {link}i
to {station} (i+1) % N;
nextstate wait;
input claim(k, t);
nextstate decision;
endstate;
state critical;
...
nextstate wait;
endstate;
state decision #unstable ;
provided i < k;
nextstate wait;
...
endstate;
endprocess;
```

Figure 8: Behavior specification of Token Ring stations.

Figure 8 presents the station specification. Let us observe the definition of the **`worried`** clock, used later in specification to detect the token loss (i.e, if nothing happens during D time units, the timeout transition **`provided worried > = D`** will be executed).

Let us also observe the output action via **`link`** signalroutes. The intent is to send the **`claim`** message to the next station, through the corresponding signalroute instance (from the **`N`** existing ones). This is done simply by using the **`i`**-th link instance, and defining the destination to be the next station **`i+1 % N`**, where **`i`** store the (integer) index of the running process. So, let us notice that the outputs encode the communication topology (here, the ring structure). That is, at the architectural level

we have only indicated the number and the type of components (e.g, `N` stations and links) and the generic connection between them (e.g, stations communicate between themselves through links).

Finally, the last point we want to stress here is the `#unstable` option attached at the `decision` state. This makes the state to be ***transient*** (invisible) at execution. From an operational point of view, a process entering an unstable state must continue immediately by firing some transition at that state and so on, until a stable state will be reached. In other words, we will consider as atomic (uninterruptible) execution steps as sequences of transitions from one stable state to another stable state.

5 Discussion

Using the CDL VERIMAG was able to model the PGM (***Pretty Good Multicast***) protocol, one of the case studies considered in the ADVANCE project (and the AGEDIS project). The starting point was an SDL specification made by France Télécom R&D. VERIMAG managed to manually ***translate*** this SDL specification into the CDL and, to improve it in several respects. For example, VERIMAG constructed an entirely parameterized specification with respect to the number of components involved in the protocol. Also, VERIMAG simplified its control structure and also the data types definition. Nevertheless, some key features are still missing from both the CDL and the new IF+TD definitions. For example, there is a limited support to describe the system topology. For example, in the token ring protocol we specify that stations may communicate ith each other through link signalroutes but nothing is said about the ***ring*** structure. This information is ***coded*** inside station definition e.g, each one computing its successor based on its process identifier. VERIMAG encountered the same problem in PGM, where the tree structure of the protocol is coded in the components definition (i.e, routers, receivers, etc.). The new public modifier for process variables makes it possible to store the system topology information in public variables, making access and topology change easier.

Another missing feature ais quantifiers (existential, universal) in expressions. Quantifiers are really powerful operators allowing to one write complex conditions easily, for example, ***if there exists some process instance such that...***, or ***if forall elements of an array holds...***, etc. At this time, we have not included them in the language for pragmatic reasons, in particular, the difficulty of evaluation at runtime. Nevertheless, we plan to include them in the final version of the language.

One of the main directions of IF2.0 evolution is to make it a basic intermediate representation language for a wide range of tools, that do not provide the simulation and analysis environment of their own. These tools may provide a mapping between their language of choice and the IF2.0 Language, thus accesing the full IF2.0 environment. At the moment we have introduced the UML style parallel/sequential states and entry/exit conditions in states to facilitate translation of UML models to the IF2.0+TD. However, this is a very limited and language specific feature.

6 Test Generation and Language Mapping Extension to IF

The IF2.0+TD Language is geared towards use for test generation purposes. Thus, it also includes a separate notation for communicating with a test generator. The language is also meant to be used as an intermediate language, different from the language of the tool, that uses the IF environment. This IF extension answers both of these needs. It is not needed for modelling, but rather for directing the work of a test generation engine and facilitating the translation of the resulting test suite back into the original tool language.

We now discuss the inputs that should be provided to the test generation tool. In addition to the IF model description itself. They consist of three distinct parts:

- the test directives;
- the test architecture;
- some mapping information, indicating the correspondence between the original language and the IF2.0 Language identifier names.

Each part is described in turn and a syntax that has been adopted for the test generator input files is proposed.

6.1 Test Directives

A test directive concentrates all the model-related information, guiding a test generator. This information includes:

- test constraints, specifying the limitations on the whole test case (start, end, include, exclude)
- coverage criteria, specifying the list of properties, to be covered by the test suite
- test purpose, specifying the test case functional behavior

The product of all the individual processes, described in the model, defines the specification state machine. A test directive can be viewed as a separate state machine, where transitions are triggered by the events that take place in the specification state machine. A test generator creates a valid test case, by exploring the legal paths, defined by the specification state machine, and using the transitions and data of these paths to stimulate the test purpose state machine. In case the path in question is accepted by the test purpose machine, it can be regarded as a valid test case, satisfying both the specification and the test purpose.

6.1.1 Test Constraints

The purpose of test constraints to give additional "global" requirements on the test selection mechanism that would be difficult (or less convenient) to express within test purposes. We propose the following test constraints:

- include: to express that at least one state of the sequences of the specification selected during test generation and leading to an accept state must satisfy the given constraint. It is similar to the "CC_some" Gotcha[**GTCB**] criteria.
- exclude: to express that no state of the sequences of the specification selected during test generation should satisfy the given constraint. It is similar to the "TC_forbidden" Gotcha constraint.
- final: to give a global constraint on the termination state of the test case. It is similar to the "TC_End" Gotcha directive.
- start: to give a global constraint on the start state of the test case. The test generator will start looking for patterns defined by the test purpose and harvest coverage information only after the specification state machine has been driven to reach the state, satisfying the start constraint.

For each of them, the constraint is expressed by a boolean predicate over a set of process variables, possibly using instance quantifications. It is not nescessary to define these variables as public in their corresponding processes in order to access them in the test directives part of the specification. Exclude constraints can also appear inside test purpose state specifications, effectively providing invariants that must hold in the corresponding states.

6.1.2 Coverage Criteria

In this release, projection coverage criteria are being proposed. They are used to generate a set of test cases with respect to a given coverage constraint. They are similar to the "CC_State_Projection" Gotcha[**GTCB**] criteria. More precisely, the constraint is expressed as a set of expressions over object variables. For each reachable combination of these expression values, a test case is generated such that at least one state of the sequences selected in the specification for test generation should satisfy the given constraint. This coverage criteria can therefore be viewed as a (possibly huge) set of ``include" constraints, the difference being that multiple test cases may be produced in the former case, not one test case, satisfying all the ``include" predicates. In addition, projection coverage criteria statement may appear inside test purpose state specifications in two different places, providing coverage invariants that must hold in the corresponding states. The first is inside the state definition, effectively directing creation of a test case per possible combination of projection variable values that must hold as long as the system remains in the specified state. The second appears in the transition guard, defining a set of test cases with the set of combinations as the corresponding transition guards.

6.1.3 Test Purpose

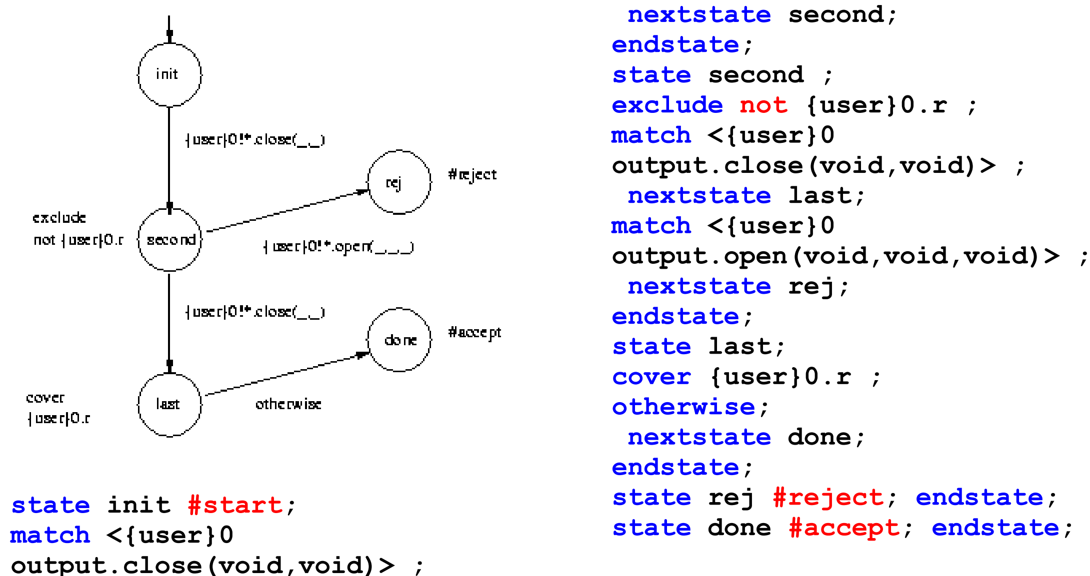
Test purposes are expressed as extended automata. They contain special states to drive the test generator when selecting the relevant execution sequences of the specification to produce the test cases: an accept state indicates a successful creation of the test case, a reject state allows to exclude some unwanted behavior, and preamble and postamble markers indicate where the complete test case should start

and terminate. Actions allowed on test purpose transitions are asynchronous emission/reception of events (expressed in the form of regular expressions) and some restricted kinds of guards over specification variables. Each test purpose transition can be taken only when its regular expression label matches the single transition, executed by the specification state machine. In addition, the test purpose transition guard expression must be satisfied. The test purpose state machine will remain in a state as long as:

- No outgoing transition from the state has a label, that matches the current specification transition, while the guard expression on the etst purpose transition is satisfied.
- The state invariant, expressed as **`exclude`** constraint, is satisfied. When the invariant doesnt hold any longer, the test purpose state machine will transition to the ***#reject*** state.

6.1.4 Test Directive Example

Let's revisit the filesystem example, discussed earlier. The test directive on figure 9 may be used to guide a test generator.




```
state init #start;
match <{user}0
output.close(void,void)> ;
 nextstate second;
endstate;
state second ;
exclude not {user}0.r ;
match <{user}0
output.close(void,void)> ;
 nextstate last;
match <{user}0
output.open(void,void,void)> ;
 nextstate rej;
endstate;
state last;
cover {user}0.r ;
otherwise;
 nextstate done;
endstate;
state rej #reject; endstate;
state done #accept; endstate;
```

Figure 9: Test Directive for the File System example.

The test directive contains a test purpose, which causes a test generator to create two test cases. We are now going to explain the meaning of individual elements on the schema of the test directive state machine. The test purpose state machine will remain in the **`init`** state, until the specification state machine performs a transition, where {user}0 sends a **`close`** signal output with arbitrary parameters (denoted by **`void`** on the test purpose transition). The test purpose state machine will then remain in the **`second`** state, until either **`close`** or **`open`** signals will be sent by the same process. The first leading to a **`last`** state, the second to a **`rej`** state. Throught the stay in the **`second`** state, the specification variable **`r`** in process {user}0 will remain true. A default transition from the sate **`last`** will be taken to the **`done`** state (since no alternative exists in this state). The coverage directive in the **`last`** state causes the creation of two

test cases, where the first satisfies the test directive with **`exclude`** {user}0.r!=true in place of coverage directive and the second one - **`exclude`** {user}0.r!=false in the same place. Coverage directives are substituted by a set of test directives, where the coverage directive in the state leads to the multitude of **`exclude`** directives. Each such directive permits only one specific combination of possible coverage expression values. The set covers all the possible value combinations.

6.2 Test Architecture

From the test generator point of view the test architecture describes the set of controllable and observable application entities (that could be either signal, method or variable names). By controllable entities we mean the ones that can be altered/called/sent by the environment to the UUT. By observable entities we understand the ones that can be examined by the environment (variable value can be checked/signal observed). These entities can be simply expressed as lists of IF2.0 identifiers (the controllable ones and the observable ones).

Observe test directives list specific observable variables/signals, out of the full list of observables, provided by the Test Architecture. These are the variables / signals, that will be checked by the test execution engine, handling the test case, that was created with these directives.

6.3 Mapping Information

This information is generated by the compiler to indicate (when necessary) the correspondence between the initial source language identifiers and the generated IF2.0 Language ones. It is used to translate the abstract test suites expressed at the IF2.0 level into the language of the original tool.

References

[AML]
The Agedis Modeling Language Specification Document, 2001.

[GTCB]
GOTCHA-TCBeans Reference Manual, Release 3.0.1, 2001.

[IF]
The IF-2.0 Language, 2001.

[TGV]
Test Generation Derived from Model Checking, Thierry J?ron and Pierre Morel, Computer Aided Verification, 1996.

A Structure

A.1 System

In general, we deal with systems composed of active objects (process instances) which are executed in parallel and interact asynchronously[1] through message (signal instances) passing via communication buffers (signalroute instances). Therefore, system specifications consist of generic components, including dynamic ones such as processes, signalroutes and signals, and static ones such as (shared) variables, data types, constant values, and external procedures.

system-decl ::=
system ***system-id*** **;**
{ ***system-component*** }*
endsystem ;

system-component ::=
process-decl |
signalroute-decl |
signal-decl |
procedure-decl |
var-decl |
type-decl |
const-decl

A.2 Process

Processes are defined as extended finite state-machines. At execution, process instances can be created and destroyed dynamically. Each process instance has a unique identifier (the pid of the instance) and an input fifo buffer storing all the incoming messages (received and not yet consumed by the instance). Process specifications include the name of the process, the number of initial instances (existing in the initial configuration of the system), the list of formal parameters (...), and various process components - the most important being the set of local variables which define the local memory and the (hierarchical) set of states (with the associated transitions) defining the behavior of the process. In addition, (local) data type, constant or procedure definitions can also be included inside [1]process specifications.

process-decl ::=
process ***process-id*** **(** ***const*** **) ;**
[fpar ***fpar-decl*** { **,** ***fpar-decl*** }* **;]**
{ ***process-component*** }*
endprocess ;

process-component ::=

[1]further development may include also synchronous interactions (e.g. rendez-vous, remote procedure calls, etc.)

state |
procedure-decl |
var-decl |
type-decl |
const-decl

A.3 Signalroute

Signalroutes denote communication paths between processes. At execution, signalroute instances can be created and destroyed dynamically. Again, each signalroute instance has a unique identifier (the pid of the instance). Signalroute specifications include the name of the signalroute, the initial number of instances, the source and destination endpoints (process or environment) and the set of signals which can be transported on it. In addition, several options are available to refine the signalroute behavior: the queuing policy (fifo or multiset), the reliability (reliable or lossy), the delivery policy ([point-to-point] peer, multicast or unicast) and the delay policy ([immediate] urgent, [overall] delay or [transmission] rate).

signalroute-decl **::=**
signalroute ***signalroute-id*** **(** ***const*** **)**
{ ***signalroute-option*** **}***
from **{** ***process-id*** | **env** **}** **to** **{** ***process-id*** | **env** **}**
with ***signal-id*** **{** **,** ***signal-id*** **}*** **;**

signalroute-option **::=**
#fifo | **#multiset** |
#reliable | **#lossy** |
#peer | **#multicast** | **#unicast** |
#urgent | **#delay[l,u]** | **#rate[l,u]**

A.4 Signal

Signals are used to communicate between processes. At execution, signal instances are sent (possibly through signalroutes) from one process instance to some other instance(s). Signal specifications include the signal name and the list of parameter types carried by the signal.

signal-decl **::=**
signal ***signal-id*** **(** **[** ***type-id*** **{** **,** ***type-id*** **}*** **]** **)** **;**

B Behavior

B.1 State

States are the main structuring concept of process behavior. In general, state specifications include the (outgoing) transitions to other states (i.e. the behavior of that state). State specifications may also include nested (sub)states which, in turn, could have their own transitions (and so on) thus refining/completing the overall

behavior of the state/process. In addition, state specifications include the set of (deferred) saved signals, which have to be postponed (if received), and the time progress condition (the temporal constraint) which limit the stay in that particular state. Options are available to define respectively the initial state of the process (or of compound states) and the stability (stable or unstable) which controls process execution steps (atomic, un-interruptable from one stable state to another). States may also include entry and exit actions (to be performed after the execution of an incoming transition or before an outgoing transition respectively). States may be defined as parallel, meaning that the process must be allowed to perform all the outgoing transitions from all the parallel active states in a single atomic transition. The sequence of these parallel ``sub-transitions'' within the atomic transition is non-deterministic. The sequencing of exit actions follows the rule: An attempt to exit a state causes an attempt to exit all the enclosed states in non-deterministic order, followed by the exit action of the state and then an attempt to exit (if necessary) the enclosing state (this attempt follows the same rule).

state ::=
state ***state-id*** { ***state-option*** }* **;**
[**tpc** ***constraint***]
[**save** ***signal-id*** { **,** ***signal-id*** }*]
[**entry** { ***statement*** }* **endentry**]
[**exit** { ***statement*** }* **endexit**]
{ ***state-component*** }*
endstate ;

state-component ::=
transition |
state

state-option ::=
#start |
#parallel |
#stable | **#unstable**

B.2 Transition

Transitions are the process reactions in response to stimulus. Transition stimuli are combinations of the following three elements : enabledness of an untimed guard (provided expression), enabledness of a timed guard (when constraint) and the presence of some signal in the input buffer of the process instance (input signal). Transition reactions are classical programs built over elementary actions using sequential composition, and conditional (if-then-else) and loop (while) control-flow statements. Elementary actions are discussed below. A transition must end either with a nextstate action (which means passing control to that state), or with a stop action (which means the destruction of the instance). In addition, each transition has a deadline {eager, delayable or lazy} denoting the priority of the transition with respect to the progress of time. Another transition property is its local priority with respect to other transitions, leading out from the same state. Input buffers to a process are fifo, which means that local priority can not cause the firing of a transition with an input other than from the first one in the buffer. However, the transition priority can

influence the choice among inputless transitions and transition with the first signal in the buffer. Priority is an integer, with 0 being the default value. Higher values mean higher priority.

```
transition ::=
[ priority integer ; ]
[ deadline { eager | delayable | lazy } ; ]
[ provided expression ; ]
[ when constraint ; ]
[ input signal-id ( [ expression { , expression }* ] ) ; ]
{ statement }*
terminator
```

```
statement ::=
action |
if expression then { statement }* [ else { statement }* ] endif |
while expression do { statement }* endwhile
```

```
terminator ::=
nextstate state-id ; |
stop ;
```

B.3 Action

The available actions which can be performed inside transitions include: skip (internal action, ignored during simulation, used for static analysis), informal (observable action, to be used to distinguish between branches of non-deterministic actions in test graphs, etc.), task (variable assignments), set (clock setting), reset (clock and variable resetting), output (signal sending, possible via signalroutes), call (procedure calls), fork (process/signalroute instance creation), kill (process/signalroute instance destruction). In particular, note that call and fork actions return values which can be used in assignments.

```
action ::=
skip ; |
informal string ; |
task expression := expression ; |
set expression := expression ; |
reset expression ; |
output signal-id ( [ expression { , expression }* ])
 [ via signalroute-id ] [ to expression ] ; |
[ expression := ] call procedure-id ( [ expression { , expression }* ] ) ; |
[ expression := ] fork process-id ( [ expression { , expression }* ] ) ; |
[ expression := ] fork signalroute-id ( [ expression { , expression }* ] ) ; |
kill expression ;
```

C Data

C.1 Constant

Constants are used to name interesting values in the system specification. Therefore, constants are used mainly in expressions, but they can also be used to define parameterized specifications through data types (e.g. the size of arrays) or system configurations (e.g. the initial number of instances). Particular constants are ***self*** which denote the pid of the instance and ***nil*** which denotes the null pid.

const-decl **::=**
const *const-id* [*const*] ;

const **::=**
true | false |
***integer* |**
***float* |**
self | nil |
const-id

C.2 Type

Five predefined types are available to the user : boolean, integer, float, pid and clock. In addition, the user may define their own data types using the standard type definition mechanisms: enumeration, range (domains), record (structures) and array. One can also define complex types using the abstract type definition mechanism - only the signature of the abstract type is defined, and in order to use it, an external implementation must be provided.

type-decl **::=**
type *type-id type* ;

type **::=**
enum *const-id* { , *const-id* }* endenum |
range *const* .. *const* |
array [*const*] of *type-id* |
record { *field-decl* }* endrecord |
string [*const*] of *type-id* |
abstract { *function-decl* }* endabstract |
type-id

field-decl **::=**
***field-id type-id* ;**

function-decl **::=**
***type-id function-id* ([*type-id* { , *type-id* }*]) ;**

C.3 Variable

Variables, including formal parameters, have name and type.

var-decl ::=
 var ***var-id type-id*** [**private** | **public**] **;**

fpar-decl ::=
 [**in** | **inout** | **out**] ***fpar-id type-id***

C.4 Expression

Expressions are built on top of constants and variables using a rich set of operators, including standard boolean operators, arithmetic operators, and comparison operators. Also, standard `.' and `[]' operators are used to access respectively fields in structured variables, or elements in arrays. In addition, we define a cast-operator allowing to convert, for instance, pid or enum values to integers (and back) and the ***active*** operator to test the activity of clocks. Finally, conditional expressions are also allowed.

expression ::=
const |
{ ***process-id*** **}** ***pid-expr*** **.** ***variable-id*** |
variable-id | ***fpar-id*** |
expression **.** ***field-id*** | ***expression*** **[** ***expression*** **]** |
function-id **(** [***expression*** { **,** ***expression*** }*] **)** |
un-op expression |
expression bin-op expression |
cast-op expression |
expression **?** ***expression*** **:** ***expression*** |
(***expression*** **)**

un-op ::=
not |
active |
+ | **–**

bin-op ::=
or | **and** |
= | **<>** | **<** | **<=** | **>=** | **>** |
+ | **–** | ***** | **/** | **%**

cast-op ::=
{ ***type-id*** | ***process-id*** | ***signalroute-id*** **}**

C.5 Constraint

Constraints are particular conditions on clocks. Basically, they are conjunctions of atomic-constraints, which are comparisons between clocks or clock differences and discrete (integer-type) expressions.

constraint ::=
atomic-constraint { **and** ***atomic-constraint*** }*

atomic-constraint **::=**
expression comp-op expression |
expression – expression comp-op expression

comp-op **::=**
= | < | < = | > = | >

D External

D.1 Procedure

Procedures are the mean to import and use external code in IF specifications. They are intended to describe intensive data transformation, executed without interaction in some instance. Procedure specifications include their signature (procedure name, formal parameters, return type) and respectively their actual code (written directly in the external language). It is also possible to use procedures, written in IF.

procedure-decl **::=**
procedure ***procedure-id*** **;**
[fpar ***fpar-decl*** **{ ,** ***fpar-decl*** **}*** **;]**
[returns ***type-id*** **;]**
[{# c code #} | { ***statement*** **}*****]**
endprocedure ;

E Test Directives

The Test Directives are the Intermediate Format Language extension for test generation purposes. Each model may contain multiple test directives. Each test directive contains all the information needed for ***one*** run of the test generation engine. This information may include many global constraints (include or exclude), one observation (observe) and one coverage criterion (cover) as well as one functional test purpose (states). Global preamble (start) and postamble (end) constraints may also be expressed, if necessary.

test-directive **::=**
[start ***bool-expression*** **;]**
[include ***bool-expression*** **, ...** ***bool-expression*** **;]**
[exclude ***bool-expression*** **, ...** ***bool-expression*** **;]**
[observe ***variable*** **, ...** ***variable***
[when ***bool-expression*** **] ;]**
[cover ***expression*** **, ...** ***expression***
[when ***bool-expression*** **] ;]**
[end ***bool-expression*** **;]**
state
...
state

States are part of the functional test purpose. Each state may add local exclude constraints, observation variables and coverage criterion.

state **::=**
state ***state-id*** **[#start | #begin | #accept | #reject | #end] ;**
[exclude ***bool-expression*** **, ...** ***bool-expression*** **;]**
[observe ***variable*** **, ...** ***variable***
[when ***bool-expression*** **] ;]**
[cover ***expression*** **, ...** ***expression***
[when ***bool-expression*** **] ;]**
transition
...
transition
[otherwise ;
nextstate ***state-id*** **;]**
endstate ;

Each transition may have an optional guard (provided), an optional matching regular expression on events (match), local coverage information (cover). Regular expressions are defined in a quite standard manner. Basic events may be qualified with the originating process identifier, which must be a pid value. Value parameters of input/output events are either explicit values or wildcards (*). The optional otherwise clause provides the default transition, taken when no match for the specification transition has been found.

During generation, specification and test purpose are strongly synchronised. Each transition taken into the specification ***must*** eventually synchronize either with some transition of the test purpose (if any), or with the ***otherwise*** clause (if present).

The meaning of an include constraint on the test directive level is to require all the boolean subexpressions to be satisfied simultaneously at some point of the test case (along the accepted path).

The meaning of an exclude constraint on the test directive level is to require that any boolean subexpression is never to be satisfied at any point of the test case (along the accepted path). This constraint can also be used on individual states, where it must hold as long as the test purpose state machine remains in that state.

The observe directive lists signals / variables that should be checked to have the predicted values during test execution. These must be listed in the observable list of the test architecture.

The projeted coverage criteria can be seen as a set of test directives, where each one requires the test case to reach a new and unique combination of covered expression values. It may appear both at the test directive level, meaning that the combination should hold throught the test case (from the start state). The coverage directive can also be linked to a specific state, which requires the combination to hold in that specific state. The directive can also appear on the guard condition, meaning that the combination of values must become part of the guard in question.

The begin and end constraints are boolean predicates on process variables. They define the start of the work part of the test case (separating possible initialization) and the final condition to be satisfied, after all the other requirements on the test case have been fulfilled.

transition ::=
[**provided** ***bool-expression*** **;**]
match ***event-regular-expression*** **;**
[**cover** ***expression*** **, ...** ***expression***
[**when** ***bool-expression***] **;**]
nextstate ***state-id*** **;**

event-regular-expression ::=
<[***pid-value*** **.**] ***event*****>** |
~ ***event-regular-expression*** |
event-regular-expression ***event-regular-expression*** |
event-regular-expression **&** ***event-regular-expression*** |
event-regular-expression **|** ***event-regular-expression*** |
event-regular-expression ***** |
event-regular-expression **+** |
(***event-regular-expression*** **)**

event ::=
any |
input ***signl-id*** **(** ***value*** | **void , ...** ***value*** | **void)** |
output ***signal-id*** **(** ***value*** | **void, ...** ***value*** | **void)** [**to** ***pid-value***] |
informal ***string***

Expressions are written using the IF-2.0 syntax. However, we require that any variable (occuring in test directives) must be explicitly qualified with the process identifier of the owner (e.g, always write ***{process-id} 3.x*** instead of ***x***. In addition, explicit quantifiers may be available soon.

Input / output events match the input by process or output by process transitions of the specification. ***any*** event matches any specification transition, while ***informal*** matches the corresponding informal action.

F Test Architecture

architecture ::=
observe
{ ***probe*** }*
control
{ ***probe*** }*

probe ::=
[***origin*** **.**] ***variable-id*** |
[***origin*** **.**] ***signal-id***

G Mapping Information

mapping **::=**
map { *map-pair* }*

map-pair **::=**
***origin*.*variable-id* -> string; |**
[*origin* .] *signal-id* -> string;

The optional mapping specification allows easy translation of the test case back to the user language.