INFORMATION SOCIETIES TECHNOLOGY (IST) PROGRAMME

Future and Emerging Technologies – Open Domain

Project acronym : ADVANCE

Project full title:

Advanced Validation Techniques
for Telecommunication Protocols

Contract no: IST-1999-29082

Deliverable D10

Integration of Abstraction Tools with Efficient Finite-State Tools

Year 2: October 2001 – September 2002

Date of preparation: October 2002

Contributors: Grenoble

Introduction

This report describes the main contributions of the Advance project concerning the integration of abstraction and model-checking tools. This integration is essentially realized through the IF-2.0 language and its environment.

Semantics

Deliverable D10/a describes the operational semantics underlying the IF-2.0 language. This semantics is taken as a reference point for integrating the tools. The key issue in defining the semantics concerns the smooth combination of process dynamic creation, handling process identifiers (pids), real-time aspects as well as asynchronous communication via timed-fifo queues.

Static Analysis revisited

Our experience with IF-1.0 has shown that simplification by means of static analysis is crucial for dealing successfully with real specifications. Moreover, even simple analysis such as live variables analysis or dead-code elimination can significantly reduce the size of the state-space of the model.

By moving up to IF-2.0, we must review all these static analysis. Several novel features of the language such as dynamic creation and destruction of processes and communication channels, plus the private/public scope of variables make them much more difficult to apply than in the previous version. In IF-1.0, specifications were completely static, the set of processes, queues, and variables is fixed and apriori known.

During this year, the following analysis have been adapted to work on IF-2.0 specifications:

Live variables

This analysis transforms an IF-2.0 specifications into an equivalent smaller one by removing globally dead variables and signal parameters and by adding systematic resets of locally dead variables. For a definition of live variables see [1].

Initially, all the private variables and signal parameters are supposed to be dead, unless otherwise specified by the user. Public variables are supposed to be always live. The analysis alternates local (standard) live variables computation on each process and inter-process live propagation through input/output signal parameters until a global fixpoint is reached.

The example below illustrates the results obtained on a version of the alternating bit protocol extended with data. The live analysis is clever enough to detect that the data variables (m) are *transparent* for the protocol (both on transmitter and receiver side) and can be eliminated. Also, some resets of control variables (b) and (c) are introduced.

```
system bitalt;

type data = range 0 .. 3;

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

signalroute et(1)
  from env to transmitter
  with put;

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

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

```
signalroute re(1)
  from receiver to env
  with get;

process transmitter(1);

var t clock;
var b boolean := false;
var c boolean;
var m data;

state idle #start;
  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;
    reset c;
      nextstate idle;
  provided c <> b;
    reset c;
      nextstate busy;
endstate;
endprocess;

process receiver(1);

var b boolean :=false;
var c boolean;
var m data;

state idle #start;
  input sdt(m, c);
    if b = c then
      output ack(b) via {rt}0;
      output get(m);
      task b := not b;
    else
      output ack(not b) via {rt}0;
    endif
    reset c;
    nextstate idle;
endstate;
endprocess;
endsystem;
```

Dead-code elimination

This analysis transforms an IF-2.0 specification by removing unreachable control states and control transitions under some user-given assumptions about the environment. More precisely, the user may indicate the set (by default empty) of signals the environment may sent to the system at execution.

This analysis solves a simple static reachability problem : it computes, for each process separately, the set of control states (transitions) which may be statically reached (executed) starting from the initial control state. Meantime, it computes the set of processes that may have running instances : either they have instances in the initial configuration or some corresponding fork action may be reached and therefore executed. When done, processes without instances are removed from the specification. The other processes are restricted to the set of reachable states and transitions.

Variable abstraction

This analysis provides a simple and efficient way of computing abstractions by eliminating variables and their dependencies from a specification. Some initial set of *abstract variables* have to be specified by the user. Then, using data and input/output dependencies, the largest set of variables which *may* be influenced by these variables is computed, at each control state. The computation proceeds as for live analysis: processes are analyzed separately, and the results obtained are propagated between them using the input/output dependencies. Finally, actions and guards involving abstract variables are removed from the specifications

Contrarily to previous analysis which are exact, variable abstraction may introduce more behaviors. Nevertheless, it always simplifies the state vector, and therefore, we can extract automatically abstracted systems allowing for symbolic verification with tools like TreX[2] or Lash[3] which make strong assumptions about the input (systems with counters, clocks, queues only). Moreover, if we are interested in explicit enumerative finite-state model checking, this technique allows to reduce the overall number of states in the semantic model.

The example below illustrates the abstraction obtained on the file system example from [5], when ones want to remove the *status* variable. The gray parts are remove from the specification.

```
process filesys(1);

type FileControlBlockType = array [NUSERS] of FileStatusType;
type SystemStatusType = array [NFILES] of FileControlBlockType;

var f FileIdType;
var u UserIdType;
var response ResponseType;
var reason ReasonType;
var systemStatus SystemStatusType;
var available boolean;

procedure File_Available_For_Read;
  fpar in f FileIdType, in u UserIdType, in systemStatus SystemStatusType;
  returns boolean;
…
endprocedure;

procedure File_Available_For_Write;
  fpar in f FileIdType, in u UserIdType, in systemStatus SystemStatusType;
  returns boolean;
…
endprocedure;


state start #start ;
    task f := 0;
    while (f < NFILES) do
      task u := 0;
      while (u < NUSERS) do
        task systemStatus[f][u] := Closed;
        task u := (u + 1);
      endwhile
      task f := (f + 1);
    endwhile
    nextstate idle;
endstate;

state idle;
  input close(f, u);
    task response := AcceptCommand;
    if (systemStatus[f][u] <> Closed) then
      task reason := FileClosed;
      task systemStatus[f][u] := Closed;
    else
      task reason := FileStatusUnchanged;
    endif
    output answer(response, reason) to ({user}u);
```

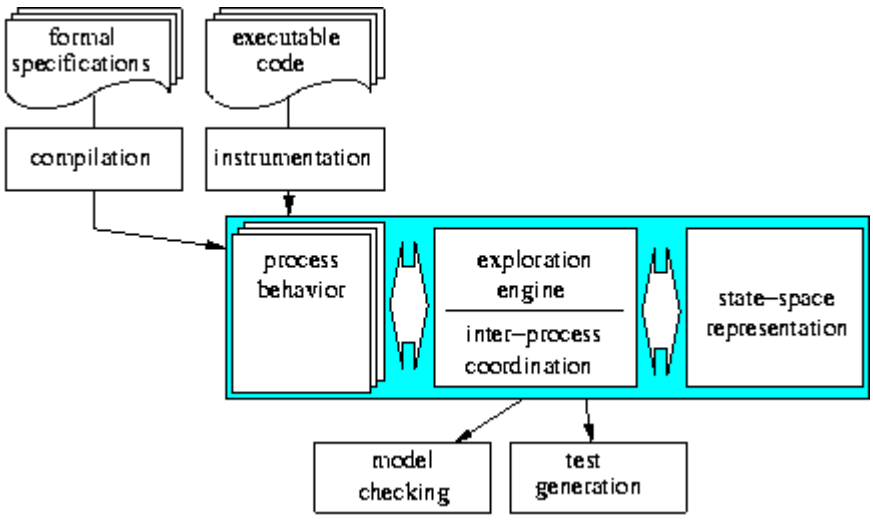
```
    nextstate idle;
  input openForRead(f, u);
    if ((systemStatus[f][u] = Reading) or
        (systemStatus[f][u] = Writing)) then
      task response := AcceptCommand;
      task reason := FileStatusUnchanged;
    else
      available := call File_Available_For_Read(f,u,systemStatus);
      if available then
        task response := AcceptCommand;
        task reason := FileOpenForRead;
        task systemStatus[f][u] := Reading;
      else
        task response := RejectCommand;
        task reason := FileLocked;
      endif
    endif
    output answer(response, reason) to ({user}u);
    nextstate idle;
  input openForWrite(f, u);
    if (systemStatus[f][u] = Writing) then
      task response := AcceptCommand;
      task reason := FileStatusUnchanged;
    else
      available := call File_Available_For_Write(f,u,systemStatus);
      if available then
        task response := AcceptCommand;
        task reason := FileOpenForWrite;
        task systemStatus[f][u] := Writing;
      else
        task response := RejectCommand;
        task reason := FileLocked;
      endif
    endif
    output answer(response, reason) to ({user}u);
    nextstate idle;
endstate;
endprocess;
```

State Space Exploration

State-space exploration is one of the successful techniques used for the analysis of concurrent systems and also the core component of any model-based validation tool (i.e. model-checker, test-generator, etc). Nevertheless, exploration is far from being trivial for dynamic systems that, in addition, use complex data, involve various communication mechanisms, mix several description languages, and moreover, depend on time constraints. The solution we propose is an *open*, *modular* and *extensible* exploration platform designed to cope with the complexity and the heterogeneity of actual concurrent systems.

The IF-2.0 exploration platform relies on a clear separation between the individual behaviour of processes and processes (i.e. memory update, transition firing) and the coordination mechanisms between processes (i.e. communication, creation, destruction). More precisely, each process or communication channel is represented as an object (in the sense of object-oriented languages) that has an internal state and may have one or more fireable (local) transitions, depending on its current state. *Time* is also a specialized process dealing with the management of all (running) clocks. Coordination is then realized by a kind of *process*

manager: it scans the set of local transitions, choose the fireable one(s) with respect to global (system) constraints, ask the corresponding processes to execute these transitions and update the global state accordingly.



This architecture provides the possibility to validate complex heterogeneous systems. Exploration is not limited to IF-2.0 specifications: any kind of processes may be run in parallel on the exploration platform as long as they implement the interface required by the process manager. It is indeed possible to use code (either directly, or instrumented accordingly) of already implemented components, instead of extracting an intermediate model to be put into some global specification.

Another advantage of the architecture is the *extensibility* concerning coordination primitives and exploration strategies. Presently, the exploration platform supports asynchronous (interleaved) execution and asynchronous point-to-point communication between processes. Different execution modes, like synchronous or run-to-completion, or additional communication mechanisms, such as broadcast or rendez-vous, simply by extending the process interfaces and the process manager functionality. Concerning the exploration strategies, reduction heuristics such as partial-order reduction or symmetry reduction are currently incorporated into the process manager. More specific heuristics may be added depending on the application domain.

In the near future, we plan to integrate the scheduling framework of [4] in order to improve the standard execution modes provided by the exploration engine (e.g. asynchronous or synchronous). Based on *dynamic priorities*, this scheduling framework is flexible and general enough to ensure a fine-grained control of execution of real-time systems, depending on various constraints. This framework fits also well in our exploration engine architecture. For instance, it is possible to extend the process manager with scheduling capabilities, in order to evaluate dynamic priorities at run-time and to restrict the set of fireable transitions accordingly.

Using Observers

The observer mechanism has been introduced in IF 2.0 to serve the following purposes:

- to provide a simple and flexible mechanism for *controlling model generation*. Thus, observers can select parts of the model to be explored, and cut off execution paths that are

irrelevant with respect to certain criteria. Consequently, they can resolve non-deterministic choices and thus act as *dynamic schedulers*. Observers can also be used to *model the reactions of a system's environment.*

- to allow *expressing (timed) linear properties* of a system in an operational way. Observers may express properties that refer both to the state of a system and to the actions performed by it.

The general idea behind observers is that their behavior is described in the same way as for the IF processes (i.e. by an extended timed state machine), but they can react "synchronously"[1] to various events and conditions occurring in the observed system.

Observation mechanisms

For monitoring system state, observers dispose of special operators for retrieving:

- values of system and process *variables*
- current *states* of the processes
- content of *queues*

For monitoring the actions performed by a system, observers dispose of constructs that retrieve *events* together with *data* relative to those events. Events are generated whenever the system executes an action such as: signal output/delivery/ input, process creation/destruction, informal statements.

At every moment, an observer can see the events that have occurred in the previous system step[1].

Expressing properties

Properties are expressed by observers by way of classifying the states into *ordinary*, *error* or *success* states. The precise semantics of this depends on the way observers are use subsequently in verification. More precisely, an observer may be used to express a *safety property*, case in which the success/error states are considered as final states of a finite automaton. Alternatively, an observer may be used to express a *liveness property*, case in which the success/error states are considered as accepting/non-accepting states of a Büchi automaton.

Observers themselves are classified into:

- *pure* - which only express a property
- *cut* - which also guide the simulation by cutting execution paths
- *intrusive* - which may also alter the system behavior (e.g. by injecting signals, etc.)

Execution model

Observers are executed in parallel with an IF specification. During model generation, the semantics is that of a "weak synchronous composition", i.e.:

- the observer receives the control after each atomic step (transition) of the system
- depending on the events and conditions occurring in the system, the observer executes 0 or more steps, in a *run-to-completion* fashion

Note that:

- an observer always watches the configuration reached by the IF system *after* the concerned system step

---

[1] See the "Execution model" subsection for the precise meaning of this.

- an observer always watches the events that have occurred during the previous system step

Example

In the following we present a small example, a property relative to a simple protocol with one *transmitter* and one *receiver*, such as the alternating bit protocol.

The property: *every time a **put(m)** message is received by the transmitter, the transmitter does not return in state **idle** until a **get(m)** with the same m is issued by the receiver.*

pure observer safety1;

```
var m data;
var n data;

state idle #start ;
    match input put(m);
      nextstate wait;
endstate;
state wait;
    provided ({transmitter}0)
        instate idle;
      nextstate err;
    match output get(n);
      nextstate dec;
endstate;
state dec #unstable ;
    provided n = m;
      nextstate idle;
    provided n <> m;
      nextstate wait;
endstate;
state err #error ;
endstate;
endobserver;
```

Model Checking Java Monitors

Deliverable D10/b describes a method for verifying monitoring properties for static Java. By static Java we mean Java programs that do not include dynamic creation.

Java contains programming primitives for synchronizing and implementing mutual exclusion of method calls. These are called Java's *monitoring primitives*. It is the programmer's responsibility to use these primitives to ensure and implement the desired synchronization and mutual exclusion properties. It's not surprising that this task is error prone.

This deliverable first formally describes the semantics of Java's monitoring primitives and then shows how using this semantics Java programs are translated to IF models. The so obtained IF models are analyzed by model checking techniques to verify the desired synchronization properties.

References

1. A. Aho, R. Sethi, J.D. Ullman. **Compilers: Principles, Techniques and Tools**. Adisson-Wesley, Readings MA, 1986
2. A. Annichini, A. Bouajjani, M. Sighireanu. **TreX : A tool for Reachability Analysis of Complex Systems**. In Proceedings of CAV'01, Paris, France. LNCS 2102
3. B. Boigelot, L. Latour. **The Liege Automata-based Symbolic Handler (LASH)**. Available at http://www.montefiore.ulg.ac.be/~boigelot/research/lash
4. K. Altisen, G. Gossler, J. Sifakis. **A methodolgy for the construction of scheduled systems**. In Proceedings of FTRTFT 2000, LNCS 1926.
5. Advance project. Deliverable D3 : **Common Model Description Language**