IF-2.0: A Validation Environment for Component-Based Real-Time Systems*

Marius Bozga, Susanne Graf, and Laurent Mounier

VERIMAG, Centre Equation, 2 avenue de Vignate, F-38610 Gières

1 Introduction

It is widely recognised that the automated validation of complex systems cannot be achieved without tool integration. The development of the IF-1.0 toolbox [3] was initiated several years ago, in order to provide an open validation platform for timed asynchronous systems (such as telecommunication protocols or distributed applications, in general). The toolbox was built upon an *intermediate representation* language based on extended timed automata. In particular, this representation allowed us to study the semantics of real-time primitives for asynchronous systems. Currently, the toolbox contains dedicated tools on the intermediate language (such as compilers, static analysers and model-checkers) as well as front-ends to various specification languages and validation tools (academic and commercial ones). Among the dedicated tools, we focused on static analysis (such as slicing and abstraction) which are mandatory for an automated validation of complex systems. Finally, the toolbox was successfully used on several case studies, the most relevant ones being presented in [4].

Despite the interest of this toolbox on specific applications, it appears that some of the initial design choices, which were made to obtain a maximal efficiency, are sometimes too restrictive. In particular they may prevent its applicability to a wider context:

- the *static* nature of the intermediate representation prevents the analysis of dynamic systems. More exactly, primitive operations like object (or thread) creation and destruction, which are widely and naturally used both in specification formalisms like UML or programming languages like Java, were not supported.
- the architecture of the exploration engine allowed only the exploration of pure IF-1.0 specifications. This is too restrictive for complex system specifications which mix formal descriptions and executable code (e.g, for components already implemented and tested).

This situation motivated the extension of the IF-1.0 intermediate representation and, in turn, to re-consider the architecture of the exploration engine. Some of the language extensions are derived from existing specification formalisms (UML [11] and

* This work was supported in part by the European Commission FET projects ADVANCE, contract No IST-1999-29082 and AGEDIS, contract No IST-1999-20218

SDL-2000 [9]) and object oriented programming languages (like Java). Concerning the exploration engine architecture, the approach we followed is influenced both by traditional model-checkers such as Spin [8] or Open/Caesar [5] and more recent run-time validation tools such as Verisoft [6], Java PathFinder [7] or SystemC [12]. The originality of this architecture is to preserve exhaustive exploration capabilities while supporting heterogeneous specifications (with external code invocations and dynamic object creations). These extensions are described in more details in the following sections, together with some running experiments and perspectives.

2 Dynamic Extended Automata

The formal basis for the IF-2.0 intermediate representation is a dynamic version of extended timed automata.

We focus on systems composed of several components (called *processes*), running in parallel and interacting through message-passing, either via communication channels (called *signalroutes*), or by direct addressing. The number of processes and signalroutes may change over time: they may be created and deleted dynamically, during the lifetime of the system.

Each process is described by an extended timed automaton. It has a unique process identifier (*pid*) value, a local memory consisting of variables (including clocks), control states and a queue of pending messages (received and not yet consumed). As usually, processes move from one control state to another by executing transitions, which are triggered by messages in the input queue and/or some (possibly timed) guards. Transition bodies are *sequential programs* consisting of elementary actions (like variable or clock assignments, message sending, process creation/destruction, external code invocation, etc) structured using elementary control-flow statements (like if-then-else, while-do, etc). Control states may be nested (as in statecharts) in order to factorize common behaviour and obtain modular automata descriptions. Signalroutes are specialised communication media that transport messages between processes. The behaviour of the signalroute is defined by its storing policy (FIFO or multiset), its delivery policy (peer to peer, unicast or multicast), its delaying policy ("zero delay", "delay" or "rate") and finally its reliability (reliable or lossy).

The semantics of the extended automata model is defined by the graph of its executions [1]. This graph is obtained by the *interleaved* execution of processes, where process transitions define *atomic* non-interruptive execution steps.

The semantics of time is similar to the one of timed automata: time progresses 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 time progress, or equivalently, the waiting time in states, we rely on transition urgencies [2] – explicit deadlines *eager*, *lazy* or *delayable* attached to transitions defining when they *must* be executed. More precisely, *eager* transitions must be executed as soon as they are enabled and waiting is not allowed; *lazy* transitions are never

[1] For pure IF-2.0 specifications there exists also a formal operational semantics, however, for specifications using external code we rely on runtime execution results.

urgent, that is, when a lazy transition is enabled the transition may be executed or, alternatively, the process may wait without any restriction; finally, when a *delayable* transition is enabled, waiting is allowed as long as time progress does not disable it.

Example 1. Consider a multi-threaded server which can handle at most `N` simultaneous requests. Thus, if possible, for a `request` message (received from the environment) a `thread` is created. The server keeps in the `thc` variable the number of running threads. Thread processes are quite simple: once created, they work, and when finished they send a `done` message back to the server. These messages are delayed through a unique signalroute `cs` (those address is passed as a parameter when creating a `thread` process).

```
signalroute cs(1) #delay[1,2]
  from thread to server
  with done;

process server(1);
    var thc integer;
  state idle #start ;
    deadline lazy;
    provided thc < N;
    input request();
      fork thread(self, cs0);
      task thc := thc + 1;
        nextstate idle;
    deadline eager;
    input done();
      task thc := thc - 1;
        nextstate idle;
  endstate;
endprocess;

process thread(0);
    fpar parent pid, route pid;
  state init #start ;
    deadline lazy;
      informal "work";
      output done()
        via route to parent;
      stop;
  endstate;
endprocess;
```

3 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 signalroute 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 specialised process dealing with the management of all (running) clocks. Coordination is then realised 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.

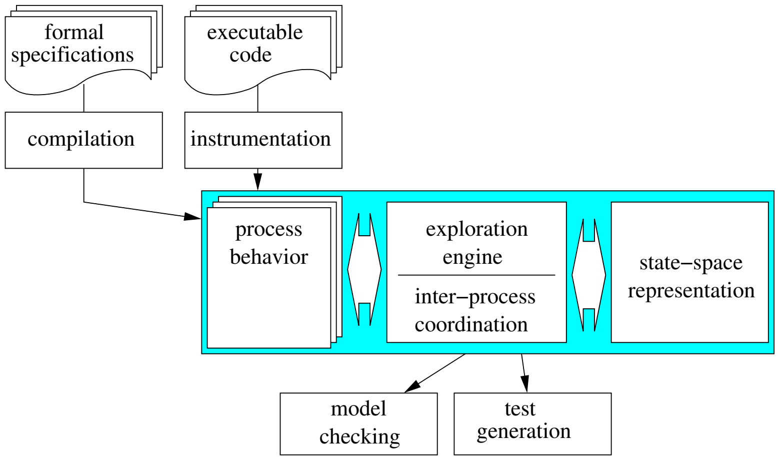

Fig. 1. Functional view of IF-2.0 exploration platform.

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.

4 Ongoing work and perspectives

The IF-2.0 representation and the associated environment are currently being used in several research projects. As example, we mention AGEDIS (see `http://www.agedis.de`) where, in cooperation with IBM and IRISA we develop a testing environment for distributed systems. In this project, IF-2.0 plays a central role, both as an (operational) representation for system's behaviour (described in UML at the user level) and as an exploration engine used by a model-based test generator (an extension of TGV [10]).

In the near future we plan to upgrade the (most effective) static analysis techniques, already implemented for IF-1.0, to the new intermediate representation IF-2.0. In particular, *slicing* and *abstraction* techniques are mandatory to keep tractable the state-space exploration. However, due to the dynamic features of IF-2.0, some of these techniques have to be revisited.

Another perspective is the integration of the scheduling framework of [1] 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.

The IF-2.0 package can be downloaded at `http://www-verimag.imag.fr/~async/IF/`.

References

1. K. Altisen, G. Gössler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *Proceedings of FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer, September 2000.
2. S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.
3. M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: A Validation Environment for Timed Asynchronous Systems. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of CAV'00 (Chicago, USA)*, volume 1855 of *LNCS*. Springer, July 2000.
4. M. Bozga, S. Graf, and L. Mounier. Automated Validation of Distributed Software using the IF Environment. In *Workshop on Software Model-Checking*, volume 55. TCS, July 2001.
5. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of TACAS'98 (Lisbon, Portugal)*, volume 1384 of *LNCS*, pages 68–84. Springer, March 1998.
6. P. Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software (short paper). In *Proceedings of CAV'97 (Haifa, Israel)*, volume 1254 of *LNCS*, pages 476–479. Springer, June 1997.
7. K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.
8. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, `http://cm.bell-labs.com/cm/cs/what/spin`, 1991.
9. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999.

10. T. Jéron and P. Morel. Test Generation Derived from Model Checking. In N. Halbwachs and D. Peled, editors, *Proceedings of CAV'99 (Trento, Italy)*, volume 1633 of *LNCS*, pages 108–122. Springer, July 1999.
11. OMG. Unified Modeling Language Specification. Technical Report OMG UML v1.3 – ad/99-06-09, Object Management Group, June 1999.
12. Stuart Swan. An Introduction to System-Level Modeling in Systemc 2.0. Technical report, Open SystemC Initiative, 2001.