Real-time Extensions for the Fiacre modeling language*

Nouha Abid Silvano Dal Zilio

CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ; F-31062 Toulouse, France
name@laas.fr

Abstract

We present our ongoing research on the extension of the Fiacre language with real-time constructs and real-time verification patterns. Fiacre is a formal language with support for expressing concurrency and timing constraints; its goal is to act as an intermediate format for the formal verification of high-level modeling language, such as Architecture Description Languages or UML profiles for system modeling. Essentially, Fiacre is designed both as the target of model transformation engines from various languages, as well as the source language of compilers into verification toolboxes, namely Tina and CADP [1]. Our motivations for extending Fiacre are to reduce the semantic gap between Fiacre and high-level description languages and to streamline our verification process.

1 Introduction

The Fiacre language has been designed in the context of the TOPCASED project [2] to serve as an intermediate format between high level description languages and verification toolboxes. The use of a formal intermediate modeling language has two benefits. First, it helps reduce the semantic gap between high-level models – expressed, for example, using Architecture Description Languages (ADL) like AADL or UML profiles for system modeling – and the input format of verification tools – that often relies on low level formalisms, such as Petri Nets or process algebra. Second, the use of a *formal language* makes it possible to define precisely the semantics of the input language "only once" and to share this work among different verification toolchains. This is particularly helpful when we try to address emergent system modeling language, whose semantic is still maturing.

We present ongoing research on the extension of the Fiacre language with real-time constructs and real-time verification patterns. Our motivations for extending Fiacre are to reduce the semantic gap between Fiacre and high-level description languages and to streamline our verification process. These extensions, based on users feedbacks, aim at integrating in the language many aspects supported by a large number of design languages and that are not yet supported by Fiacre.

In Section 2, we present the context of my research and briefly define the Fiacre language using an illustrative example. In Section 3 we details some proposed extensions and some preliminary result.

2 Context of the Research

Formal verification is advocated as one of the solutions to the consistent increase in design complexity of real-time embedded software. While verification activities should be performed at all stages of the

*This work is partly supported by the FNRAE project Quarteft and by region Midi-Pyrénées.

development process to ensure the quality and reliability of systems, there are strong incentives for systems designers to carry out as much verification as possible during the early phases, especially during the functional and architectural design phases. To support this trend, a number of high level system modeling languages have been proposed. To support model verification, a specific verification toolchain should be developed for each of these modeling languages – even when they share strong commonalities. The Fiacre language offers a solution to simplify the development of these verification toolchains.

Fiacre is a (French) acronym for an *Intermediate Format for Embedded Distributed Components Architectures* (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués). It is a formal specification languages to represent both the behavioral and timing aspects of real-time systems. We give an example of Fiacre syntax in Listing 1 and 2 of this paper.

The Fiacre language is stratified in two syntactical categories and embeds the notions of:

Processes – that describes the behavior of sequential components. A process is defined by a set of control states, each associated with an expression that specifies state transitions. Expressions are built from deterministic constructs available in classical programming languages (assignments, conditionals, sequential composition, ...); non-deterministic constructs (choice and non-deterministic assignments); communication events on ports; and jump to next state.

Components – that describes the composition of processes, possibly in a hierarchical manner. A component is defined as a parallel composition of components and processes communicating through ports and shared variables. The notion of component allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints with communications and to define priority between communication events.

Fiacre supports two of the most common communication paradigms: communication through shared variable and synchronization through (synchronous) communication ports. In the latter case, it is possible to associate time and priority constraints to communication over ports. The ability to directly express timing constraints in a program is one of the distinguishing features of the Fiacre language.

The use of timing constraints is illustrated in the example bellow which models the operation of a simple manufacturing plant. A *factory* assembles products from two *command lines*, L1 and L2. There are four kind of machines available M1, M2, M3 and M4. L1 uses machines M1, M2 and M3 in this order while L2 uses machines M2, M3 and M4. Two *workers* and one *technician* operate the lines: worker W1 is on line L1; worker W2 on line L2; and technician T1 maintains the machines on both lines. The factory is subject to operational and legal requirements: (1) workers should operate on work cycles of less than 35 minutes, separated by 5 minutes pauses; (2) the duration of a machine task is between 5 and 10 minutes; (3) machines should be maintained after 15 task cycles.

The Fiacre language is strongly typed. We start the specification with the declaration of the types used in our model of the plant. We use *union types* to model enumerations but more complex structured types are also supported. The types used in the model of the plant are :

type machinename is union $M1|M2|M3|M4$ end

type linename is union $line1|line2$ end

type vector is array 3 of machinename

type tab is array 2 of linename

Processes and components declarations follow. Each process declaration states the name of the process; the list of its communication ports (between brackets); the list of its parameters (between parentheses); and its list of states, after the `states` keyword. A process declaration describes the possible

transitions from each of its state, introduced by the keyword from. We present in Listing 2 part of the manufacturing plant : the machine and the worker process.

Listing 1: A factory example in Fiacre : processes

```
process Machine [StartMachine: machinename, EndMachine: machinename,
                 StartMachineMnt: machinename, EndMachineMnt: machinename]
                 (name: machinename) is

   states Idle, Work, Mnt

   var x: linename, y: machinename, cycle: nat:=0

   from Idle
   StartMachine?name;
   to Work

   from Work
   if cycle <= 15 then
   EndMachine!name; cycle:= cycle+1; to Idle
   else
   StartMachineMnt!name; to Mnt
   end

   from Mnt
   EndMachineMnt?y where y= name; cycle:=0;
   to Idle


process Worker[Startline: linename, Endline: linename] (line: linename) is

    states Idle, Work, Pause

    var y: linename

    from Idle
    Startline?y where y= line;
    to Work

    from Work
    Endline?y where y=line;
    to Pause

    from Pause
    wait [5,5];
    to Idle
```

A component declaration describes how process instances interact as presented in Listing 3. For instance, communications on ports can be synchronized or interleaved.

Listing 2: A factory example in Fiacre : components

```
component Work [ Startline: linename, Endline: linename, StartMachine:
                 machinename, EndMachine: machinename] (line: linename) is

             par * in
                 Worker[Startline, Endline] (line)
              || L1 [Startline, StartMachine, EndMachine, Endline]
             end


component Main is

     port   Startline: linename in [0,0],
```

```
                Endline: linename in [0,0],
                StartMachine: machinename in [0,0],
                EndMachine: machinename in [5,10],
                observer: none in [35,35],
                StartMachineMnt: machinename,
                EndMachineMnt: machinename in [0,5]

        par * in
              P1 [Startline, Endline]
             || par
                  Work [Startline, Endline, StartMachine, EndMachine] (line1)
                || Work [Startline, Endline, StartMachine, EndMachine] (line2)
                end
             || par * in
                Technician  [StartMachineMnt, EndMachineMnt]
                ||
                par M1 [StartMachine, EndMachine, StartMachineMnt, EndMachineMnt]
                 || M2 [StartMachine, EndMachine, StartMachineMnt, EndMachineMnt]
                 || M3 [StartMachine, EndMachine, StartMachineMnt, EndMachineMnt]
                 || M4 [StartMachine, EndMachine, StartMachineMnt, EndMachineMnt]
                end
             end
         end
```

The " meaning" of a Fiacre program can be expressed as a Timed Transition System, defined from the states of the system processes and from timed transitions between these states. The *frac* compiler can be used to build a Time Transition System (TTS) from a Fiacre program. The Tina verification toolbox [3] (`http://homepages.laas.fr/bernard/tina`) offers several tools to work with TTS files[1]. For instance, for verification purposes, TTS specifications can be used by *selt* – a model-checker for a State-Event version of LTL – and by *muse* – a model-checker for the μ-calculus. A verification toolchain from Fiacre to CADP is also available.

We can express (a very weak form of) the real-time requirements of the factory using LTL formulas. For instance:

Property 1 If Worker 1 is in a work cycle (in the state `Work` of instance `1` of the `Worker` process) then he will eventually rests (he will reach the state `Pause`): `Worker_1_sWork => <>Worker_1_sPause`

Property 2 Machine 1 should always be eventually maintained:
`[](<>(Machine_1_sWork => Machine_1_sMnt))`

A strong limitation of an approach based on LTL model-checking is that it is not possible to express timing constraints like, for example, that some deadline between significant events are met. In the following section, we show some extensions to the Fiacre language that makes it easier to express timed properties on systems (instead of simple temporal properties).

3 Real-Time Extensions to Fiacre: the RT-Fiacre language

In this section, we discuss possible extensions to Fiacre in order to consider real time aspects. We consider two kinds of extensions: (1) *behavioral extensions*, with the aim to increase the expressivity or enhance the "ease of description" of the language; and (2) *properties extensions*, that aim at extending the set of properties that can be checked on a model. The language with his extensions should be called RT-Fiacre. We give one example for each kind of extensions.

[1] Tina is primarily a toolbox for the edition and analysis of Petri Nets and Time Petri Nets.

3.1 Behavioral Extensions

We collected feedbacks from users that applied Fiacre to translate high-level specification languages. A common limitation expressed by users was the lack of constructs found in real-time languages, and most particularly the absence of the notion of "scheduled task". Actually, Fiacre does not impose a specific scheduling policy for managing processes – or a high-level mean to express such policy – which means that users "should write their scheduler". To this end, among the proposed behavioral extensions, we can point out the addition of support for declaring sporadic and periodic tasks. For example, future versions of RT-Fiacre will include a specific declaration for real-time processes with a syntax of the form:

```
periodic(period=p, deadline=d) process P [c1,...,cn] ...
```

for (preemptible) processes that should be scheduled every `p` units of time and:

```
sporadic(period=p, deadline=d, priority=p, protocol=proto) ...
```

for sporadic processes that can appear every `p` units of time.

3.2 Properties Extensions

We have given a brief idea of how Fiacre can be used for the model verification of high-level description languages. Models are first interpreted in Fiacre using a dedicated translation tool (one compiler per high-level language) and then compiled and checked using a model checker. We already gave some examples of LTL formulas that can be checked by Tina on the specification given in Listing 1. When working with industrial partners, it is often not desirable to expose this level of details. Instead, we propose a set of high-level *verification patterns* that are rich enough to express (the most common) user requirements. Hence, RT-Fiacre will integrate a "property declaration" language with high-level requirement pattern such as, for example, the property *Event1* `leadsto` *Event2* `within` *I*, where *I* is a time interval, meaning that: whenever *Event1* occurs, then eventually *Event2* must occur in a delay bounded by *I*. Events that are observable at the Fiacre level are: a process entering or leaving a state; a variable changing value; a communication through a port. We can exemplify the use of the *leadsto* pattern on the factory example. For instance, the requirement that each Worker must pause after at most 35 minutes of work could be "integrated" in a RT-Fiacre program by including a property declaration of the form:

```
Worker_1_sWork leadsto Worker_1_sPause within [0, 35]
```

We have already implemented an extension to the frac compiler that accepts the declaration of verification pattern. Currently, timed patterns, such as the "leadsto property", are compiled into an oberver that is automatically composed with the system at the level of the Timed Transition System. In the case where the pattern is not valid, we obtain a counter-example, that is a sequence of events (with time information) that leads to a problematic state.

4 Conclusion and Perspectives

We have briefly sketched the context of the work that should be tackled during my PhD thesis entitled "Real-time extensions for the Fiacre language". While, until now, the work has mainly focused on the definition and verification of *timed verification patterns* for real-time systems, we are currently investigating behavioral extensions to the language (see Section 3.1 for an example). Together with the addition of support for scheduled task, we are also interested by adding support for the *the notion of modes*, that is often found in real-time system. We plan to base our proposal on the "mode mechanism" found in Architecture Description Languages, such as AADL for example.

References

[1] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang and F. Vernadat. Fiacre: an intermediate language for model verification in the TOPCASED environment. *4th European Congress Embedded Real Time Software* , 2008

[2] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crgut, M. Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. *3th European Congress Embedded Real Time Software*, 2006.

[3] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.