Modeling Embedded Real-Time Applications with Objects and Events

Volkan Arslan[1], Patrick Eugster[2], Piotr Nienaltowski[1]
[1]Chair of Software Engineering, ETH Zurich, 8092 Zurich, Switzerland
[2]Dept. of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA
Volkan.Arslan@inf.ethz.ch, p@cs.purdue.edu, Piotr.Nienaltowski@inf.ethz.ch

Abstract

The ability to model periodic, sporadic and aperiodic tasks in a way that ensures their timing constraints such as worst-case execution time, deadline and periodicity is a major concern in embedded real-time programming. We propose the use of a concurrent event library to achieve the predictability of embedded real-time programs while retaining the advantages of modular development and reasoning of object-oriented languages and the benefit of event-driven programming.

1. Introduction

According to [1], a *real-time system* is any system that has to respond to externally generated input stimuli (including the passage of time) within a finite and specified time interval. This means that the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced. In particular, a correct but late response is as bad as a wrong response. Hence *predictability* is a key requirement for real-time applications.

This paper presents an object-oriented event library for real-time programming, dubbed RTEL (real-time event library). RTEL is built on top of an existing event library[1] designed initially without real-time concerns in mind, and combines the power of the object paradigm (modular reasoning and development, fostering reusability and extendibility) with the benefits of events (separation of concerns by distinguishing between *application layer* (business logic) that provides operations to execute, and the *presentation layer* (user interface) that triggers the execution of these operations in response to human users' actions [2]).

To achieve predictability, RTEL is designed to support periodic, sporadic, and aperiodic tasks and to ensure their timing constraints such as worst-case execution time (WCET), deadline, and periodicity.

The rest of this paper is organized as follows: Section 2 explains how to use RTEL by means of a detailed example. Section 3 describes the architecture of RTEL and discusses some modeling aspects. Section 4 draws conclusions and discusses possible extensions of our event-driven approach to embedded real-time programming.

[1] Available for download at http://se.inf.ethz.ch/people/arslan/

2. Presentation by Example

Following [2] we use a small sample application to show the basic capabilities of RTEL for embedded and real-time programming. All code samples below use Eiffel notation [6].

In our application we want to observe the temperature, humidity, and pressure of containers in a chemical plant. The measurements are supposed to originate from external physical sensors. Whenever the value(s) of one or more measured physical attributes change(s), the concerned parts of our system (e.g. an actuator or display units) are notified, so that they can update the values and take appropriate actions.

An event-based architecture offers several benefits for such applications. First, the event-driven nature of the problem is taken into account: input values are coming from external sensors at unpredictable moments, and the application is reacting to their change. Second, we are able to preserve the independence between the *application layer* (actuators) and the *presentation layer* (UI): if the physical setup changes (e.g. sensors are replaced by different ones, actuators are changed or extended, new display units are introduced), the system can be easily adapted without the need to rewrite the application.

Now assume that we would like to control a certain actuator, e.g. a valve that adjusts the heater according to temperature changes. Since we are interested in checking the temperature regularly, we model it using a *periodic* task (i.e. a real-time task which is activated regularly at fixed rates/periods [4]). In order to model a periodic task, the timing constraints periodicity *T*, the deadline *d* and the WCET *wcet* must be specified. The deadline *d* is the point in time at which a real-time task must be completed. The WCET is the maximum amount of time needed to finish a task.

We would like to check the temperature and adjust the valves every 20 ms, and the total time needed for reading the temperature and adjusting the valve is at most 15 ms. Therefore, we take T = 20 ms and d = 15 ms.

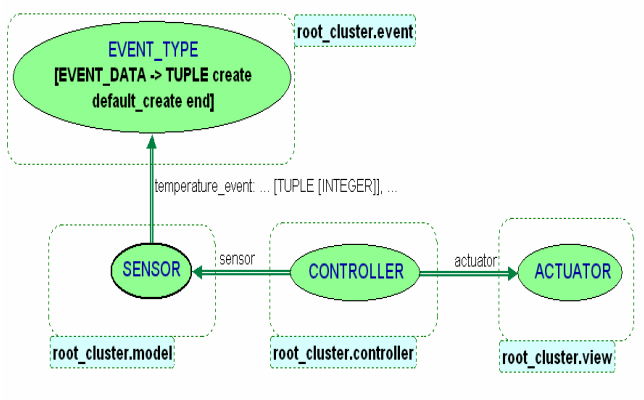

Fig. 1. Class diagram of the sample application

Figure 1 shows the overall architecture of our sample application using the BON notation [5]. Ellipses represent classes; arrows represent client-supplier relationship.

The application is divided into four clusters[1]: *event*, *model*, *controller*, and *view*. The *event* cluster contains class *EVENT_TYPE* that abstracts the general notion of event type [2], and the class *TIMED_ EVENT_TYPE*. The *model* cluster contains the application-specific classes such as *SENSOR*. The *view* cluster regroups classes which are related to the interaction with physical devices such as actuators controlling valves of the heater. Finally, class *CONTROLLER* connects classes from *model* and *view* clusters following the standard model-view-controller (MVC) design pattern.

In order to model a periodic task we declare an entity *periodical_event* as

periodical_event: **separate** *TIMED_EVENT_TYPE* [*TUPLE* [*ANY*]]

Class *TIMED_EVENT_TYPE* is an extension of class *EVENT_TYPE* [2]. It relies on the SCOOP concurrency model [7]. Keyword **separate** reflects the concurrent nature of *periodical_event*. The object represented by *periodical_event* is handled by a different processor[2] than the object that declares *periodical_event*. As a consequence, all feature calls on *periodical_event* will be executed asynchronously. The generic parameter *TUPLE* [*ANY*] represents the event data. The class in which the declaration of *periodical_event* takes place is called the *publisher* [2].

An object of type *TIMED_EVENT_TYPE* can be created and attached to *periodical_event*

create *periodical_event*

By calling the feature *publish_periodically* as in

periodical_event.publish_periodically (20)

the event *periodical_event* will be published every 20 ms. The actual publication of the event will be taken care of by the runtime system of RTEL.

It is necessary to subscribe a certain feature of a class to the event *periodical_event*, so that whenever the event is published (fired), the subscribed feature will be executed. The object which is in charge of subscribing a feature is called the *subscriber*, and the object whose feature is subscribed is called the *subscribed object*.

Consider the following feature from class *PHYSICAL_SENSOR*:

```
class PHYSICAL_SENSOR
feature -- Basic operation
 read_and_set_temperature is
      -- Read temperature of container and
      -- set the temperature of sensor object.
   do
      temperature := …
      sensor.set_temperature (temperature)
   ensure
      wcet (5)
   end
feature {NONE} -- Implementation
  temperature: INTEGER is
  sensor: separate SENSOR
end -- class PHYSICAL_SENSOR
```

Note the use of postcondition to specify the WCET. In this particular case, we set it to 5 ms. Assume that we want to subscribe this feature to *periodical_event*. This is how the subscription can be performed (in class *CONTROLLER*, which is the subscriber):

```
physical_sensor: PHYSICAL_SENSOR
…
if periodical_event.is_schedulable
  (agent physical_sensor.read_and_set_temperature, 15)
then periodical_event.subscribe
      (agent  physical_sensor.read_and_set_temperature)
end
```

We use so-called agents [6] to wrap routine calls. **agent** *x.f* (*a*) denotes an object representing the operation *x.f* (*a*). We first check if **agent** *physical_sensor.read_*

[1] In Eiffel a *cluster* is a coherent group of classes. Clusters can be hierarchical; here the root cluster is called *root_cluster*. With *event* being defined in that cluster, its full name becomes *root_cluster.event*.

[2] A processor is an abstract notion of an autonomous thread of control for the sequential execution of instructions on one or more objects. It may be implemented, e.g. as an OS process or a single thread. Current implementation of SCOOP (available at http://se.inf.ethz.ch/download) maps processors to POSIX or .NET threads.

and_set_temperature is schedulable given the timing constraints *d* = 15 and *wcet* = 5 (the latter is extracted from the postcondition of *read_and_set_temperature*). The subscription only takes place if the agent is schedulable.

Let us now consider the remaining functionality of our application. We will focus on classes *SENSOR* and *ACTUATOR*. Class *SENSOR* is an abstraction of a sensor that measures among others the temperature:

```
class SENSOR
feature -- Access
 temperature: INTEGER
 set_temperature (t: INTEGER) is
     -- Set temperature to t.
   require
     valid_temperature: t > -100 and t < 1000
   do
     temperature := t
     temperature_event.publish ([temperature])
   ensure
     temperature_set: temperature = t
   end
feature -- Events
 temperature_event: separate EVENT_TYPE
                              [TUPLE [INTEGER]]
end -- class SENSOR
```

The type of *temperature_event* is based on generic class *EVENT_TYPE*. This class takes a generic parameter *EVENT_ DATA* that represents a tuple of arbitrary types. In the case of *temperature_event*, the value of this generic parameter is *TUPLE* [*INTEGER*] since the actual event data (i.e., temperature value) is of type *INTEGER*. See [2] for a more detailed discussion on *EVENT_TYPE*.

We introduce class *ACTUATOR* that provides feature *adjust_valve*:

```
class ACTUATOR
feature -- Basic operations
 adjust_valve (t: INTEGER) is
     -- Adjust the heater accordingly.
   do
     …
   ensure
    wcet (5)
   end
  …
end -- class ACTUATOR
```

CONTROLLER subscribes feature *adjust_valve* to the corresponding event type *temperature_event*

```
   actuator: ACTUATOR:
   …
   sensor.temperature_event.subscribe
       (agent actuator.adjust_valve (?))
```

As a result, feature *adjust_valve* of *actuator* will be called each time *temperature_event* is published. The actual argument of feature *subscribe* is an agent. The question mark reflects an open argument that will be filled with concrete event data (here, an *INTEGER* value) when feature *adjust_valve* is actually executed [6].

Let us summarize: feature *read_and_set_temperature* will be executed every 20 ms as a result of *periodical_event*; its execution will cause the publication of *temperature_event* (as defined in class *SENSOR*). As a reaction to that event, the subscribed feature *actuator.adjust_valve* will be executed. As a result, every 20 ms the temperature will be read and the valves will be adjusted accordingly, which is precisely the purpose of our little application.

3. Design and Architecture of RTEL

In this section, we give an overview of RTEL. The logical components of the event library such as *event type*, *event*, *publisher*, *subscriber*, and *subscribed object* are elaborated in detail in [2].

3.1 Space Decoupling

RTEL provides *space decoupling* [3]. This important feature is neglected by many implementations of the MVC design pattern in mainstream object-oriented languages. In our example, space decoupling manifests itself in that that the *publisher* (instance of class *SENSOR*) and the *subscribed object* (instance of class *ACTUATOR*) do not know each other, i.e. there is neither a client-supplier nor an inheritance relationship between them (in Figure 1). Hence, both classes can be extended independently. For example, extending class *SENSOR* only impacts the subscriber whose role consists in connecting the publisher and the subscribed object. As a consequence, only class *CONTROLLER* needs to be extended if we use some descendant classes of *SENSOR* and *ACTUATOR*. In the basic MVC pattern, in contrast, the *view* directly invokes methods from the *model* (see e.g. [8]).

3.2 Flow Decoupling

Similarly, RTEL provides *flow decoupling* [3] thanks to the use of SCOOP – the publisher is not blocked while publishing the event *temperature_event* because *temperature_event* is declared as **separate** and therefore the call on *temperature_event* is asynchronous. Likewise, the subscribed object (instance of *ACTUATOR*) is not actively *pulling* for events. Thereby, all involved objects (instances of *SENSOR*, *ACTUATOR*, *CONTROLLER*,

EVENT_TYPE, and *TIMED_EVENT_TYPE*) are separate, i.e. they are handled by different processors.

3.3 Ensuring Real-Time Predictability

The RTEL library we proposed relies conceptually on a time-triggered scheduler (e.g. "earliest-deadline-first"-scheduler such as implemented in the RTOS XO/2 [4]) to ensure predictability, although other scheduling policies such as Rate Monotonic could be used. Since timing constraints such as WCET, periodicity, and deadline are specified, the time-triggered scheduler can decide if it can fulfill the timing constraints of the real-time task (feature) subscribed to a particular event type (such as *periodic_event* in our example). If this is not the case, the subscription of that particular task will be rejected. Similarly, *sporadic* real-time tasks can be modeled by specifying the *minimum interarrival time* instead of periodicity. Specification of aperiodic real-time tasks is much more challenging. One workaround widely applied in practice consists in *mapping* aperiodic tasks to periodic tasks through various techniques if predictability is an absolute requirement.

3.4 Modeling Aspects

The major advantage of using an event-driven approach is the separation of concerns. Logically, features *read_and_set_temperature* and *adjust_valve* are absolutely independent from each other. Both operations are part of different abstractions, and therefore they are modeled in two different classes (*PHYSICAL_SENSOR* and *ACTUATOR*, respectively). It is just in a particular constellation that both operations are *connected* together (in class *CONTROLLER*) via an event to achieve a higher goal, but enriched with additional properties such as being periodic.

Compared to traditional real-time modeling (including more recent approaches, e.g., [9]), where both operations *read_and_set_temperature* and *adjust_valve* would be defined in the body of an infinite loop of a real-time thread, the event-driven approach provides more flexibility. The *gluing* of both operations is done by the subscriber whose role is precisely to connect two different abstractions and to add further functionality. Also, timing constraints are either part of class interfaces or they are passed as feature arguments, which allows for a simple redefinition and extension of the system. In traditional approaches, on the other hand, timing constraints are *hardcoded*, which makes any extension extremely difficult.

4. Conclusions and Ongoing Work

We presented a real-time event library (RTEL) combining the benefits of object-oriented and event-driven programming to facilitate the modeling of real-time applications. We used a sample application to demonstrate the basic features of RTEL. RTEL fully adheres to the principles of object-oriented programming and makes extensive use of advanced mechanisms such as genericity, agents, and SCOOP. A salient difference with respect to mainstream real-time frameworks is that timing constraints are not scattered over distinct logical components. At the same time our approach does not rely on specific language constructs (cf. [10]). It should be noted that the RTEL is conceptually platform independent. The only requirement of the RTEL is that the underlying RTOS provides a time-triggered scheduler.

We are currently in the process of integrating RTEL with a suitable time-triggered scheduler. We are also investigating the feasibility of providing specific support for aperiodic real-time tasks.

References

[1] S. Young, Real Time Languages: Design and Development, *Ellis Horwood Publishers*, Chichester, 1982

[2] Volkan Arslan, Piotr Nienaltowski, Karine Arnout: Event Library: an object-oriented library for event-driven design, In: *Joint Modular Languages Conference (JMLC) 2003*, Klagenfurt, Austria, August 25-27, 2003

[3] P. Th. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec: The Many Faces of Publish/Subscribe, In: *ACM Comput. Surv.*, 35 (12), pages 114 − 131, 2003.

[4] Roberto Brega: A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications, *PhD thesis, ETH Zurich,* Switzerland, 2002

[5] Kim Walden, Jean-Marc Nerson: Seamless object-oriented software architecture, *Prentice Hall*, 1995.

[6] Eiffel ECMA Standard - 367: Eiffel Analysis, Design and Programming Language, available for download at http://www.ecma-international.org/

[7] Piotr Nienaltowski, Volkan Arslan, Bertrand Meyer: Concurrent object-oriented programming on .NET, In: *IEE Proceedings Software, Special Issue on ROTOR*, October 2003.

[8] Model-View-Controller, In: *Java BluePrints*, available for download at http://java.sun.com/blueprints/ patterns/MVC-detailed.html

[9] Andy Wellings, Concurrent and Real-Time Programming in Java, *John Wiley & Sons* England, 2004

[10] B. Nielsen, G. Agha: Towards Reusable Real-Time Objects. In: *Annals of Software Engineering: Special Volume on Real-Time Software Engineering*, vol. 7, pp 257-282, 1999.