GENERIC REAL-TIME INFRASTRUCTURE FOR SIGNAL ACQUISITION, GENERATION AND PROCESSING

Herman Bruyninckx, **Peter Soetens**
Dept. of Mechanical Engineering, K.U. Leuven, Belgium
`http://www.orocos.org`
`(herman.bruyninckx)/(peter.soetens)@mech.kuleuven.ac.be`

Abstract

This paper presents the design for the real-time core of a generic real-time signal acquisition, processing and generation system. The design is generic in the sense that it can be used in a wide variation of application domains, from the PLC logic of MatPLC, [7], over medical signal processing in the RT-Lab project, [2], to the robot motion control of the Orocos project, [1]. It's exactly the functional overlap between these projects that has stimulated the presented design, in the hope of gaining more critical mass for the real-time signal processing core of these, and other, projects. The paper only presents the design of an application-independent real-time signal processing infrastructure, and its implementation on a Linux RTOS, but not the protocols and plug-in functionality to be provided by the applications, on top of this infrastructure.

1 Introduction

This text presents the design for a generic real-time signal processing system. This introduction gives an overview of the design, the envisaged applications, and the specifications for its components. The presented design for a generic real-time signal processing system is "generic" because it covers the needs of several application areas:

- pure data acquisition, as implemented by Comedi, [6]. (This design can be seen as an extension to Comedi; in any case, a seemless integration with Comedi is a key specification.)
- extended data acquisition and generation, with pre- and post-processing of the signals. E.g., applications which must calibrate equipment against standards, send specific pulse trains, detect peaks and abrupt changes, estimate indirectly observed system parameters, etc.
- feedforward/feedback control, such as in robotics, or other mechatronic systems.

The presented design is limited to the *common real-time infrastructure* needed by all these applications. Application-specific functionality must be implemented on top of it, via a *"plug-in"* architecture. The presented design is also generic in a more technical sense, in that it is only *loosely coupled* to the RTOS it runs on:

- It requires only very minimal support of the RTOS, and this support is *localized* in only a few parts of the system.
- It doesn't rely at all on the real-time scheduling behaviour of the RTOS.
- It relies on clearly marked and specifief stubs to the hardware and RTOS, such that it can be ported to user space, for example for simulation or non-real-time signal processing.

We call the presented structure the *Software Pattern* [4] of real-time control: it is the structure that has proven over the last 50 years to successfully cope with all control and signal processing applications. This paper makes this Software Pattern explicit, and explains its application in various areas. It also presents the Orocos *Framework* [3] that implements this Software Pattern.

2 Components

This Section presents the components that have to be available in the system, and describes their functionalities. Figure 1 gives an overview of the components and their interaction.

2.1 Functional components

In its envisaged applications, the signal generation and processing takes place in hard real-time, but a different, time-varying frequency could be used for each of its constituent *components*:

- *Scanner*: measures signals on interface cards.
- *Actuator*: applies setpoints to interface cards.

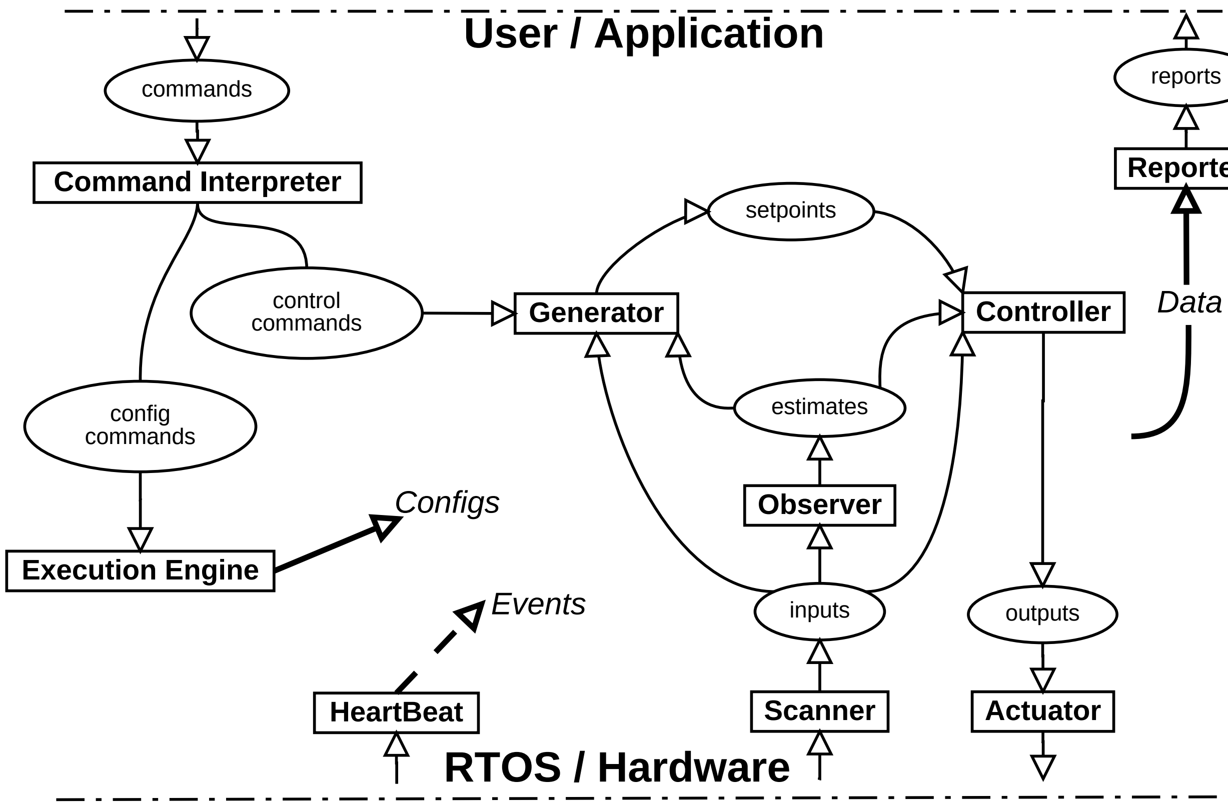

Figure 1: Structure of the real-time core components.

- *Generator*: generates signal setpoints. It supports *hybrid signals*, i.e., discrete signals ("pulses"), analog signals (in sampled form, of course), as well as discrete switches between analog signal forms. In its signal generation, it can make use of the data that other components have available.

  In control theory, one calls the functionality offered by the Generator "*feedforward*" and/or "*setpoint generation.*"

- *Observer*: reads Generator and Scanner results, and calculates estimates on these data. Lots of application-dependent forms of data observation exist, known under names such as "filtering," "transformations," "data reduction," "classification," etc.

- *Controller*: reads Generator, Scanner and Observer, and calculates setpoints for the Actuator, in its "*control algorithm.*"

These are the *functional components*, i.e., the components of which application programmers see the plug-in interface, and for which they must provide functional contents, in the form of the signal generation or processing algorithms of their application.

2.2 Infrastructural components

The design also needs some *infrastructural components*, that run "behind the screens" in order to support the functional components:

- *Execution Engine*: is responsible for *activation* and *configuration*:

  - activating the functional components respecting their individual timing specifications.
  - run-time configuration of the functional components.

  This is the only component that knows how the other components should interact, and it triggers other components to execute their functionality. By localizing the *application logic* in one single component, the system is much easier to understand, program, adapt, and make deterministic. The core of the Execution Engine is a finite state machine (or any other "decision making" engine that the application builder wants to use), whose outputs are triggers ("events") for the other components; for the pacing of its state machine, the Execution Engine relies on basic timer functionality of the

(RT)OS, but only indirectly (see *HeartBeat* below).

- *Command Interpreter*: this is *not* a hard real-time component, because it receives commands (configuration, action specification, etc.) from *user space* (in whatever *protocol* the application uses), parses them, checks their consistency, fills in the configuration data structures for the other components, and signals the Execution Engine when a complete and consistent new specification for the real-time system is available. It has to make sure that its communication with the real-time Execution Engine is *atomic*: either the whole new specification is transferred, or nothing. "*Swinging buffers*" are a possible RTOS IPC primitive to implement this atomicity, but certainly not the only one.

- *Reporter*: collects the data that the other components want to send to the user, and takes care of the transmission.

- *HeartBeat*: fires events (Sect. 3.2) at specified times, in order to "schedule" the activities of the other components in the framework. The time can be *virtual*, which is useful for scaling up or down an application's execution, and even mandatory for a distributed application, [5]. The Execution Engine configures the HeartBeat, because it knows what must be done at any given time in the application.

Only the Scanner and Actuator interact with the hardware, or rather, with a device driver library such as Comedi. Only the Reporter and the Command Interpreter interact with the user space application. Only the Execution Engine interacts with the operating system. (Also the device driver library interacts with the hardware and the operating system, but this is not visible anymore at the interface that Scanner and Actuator use.) Generator, Observer and Controller can be designed fully independently of RTOS and user, because they interact only with Scanner, Actuator, Reporter and Execution Engine.

2.3 Using the components

Application programmers don't have to use all functional components. For example, one application needs only signal generation; another one monitors an incoming signal and executes an alarm when certain signal properties are observed; etc. For every components they do use, they have to provide a *plug-in* function (Section 5). That is, each component has a number of data structures and method calls, whose existence and programming interface is defined by the design, but whose concrete implementation depends on the application. The specification of this user-interaction API (Application Programming Interface) and message protocols, however, is beyond the scope of this paper.

2.4 Flexibility

The system must be flexible, i.e., it must allow different ways to map the above-mentioned architecture onto the hardware to be used:

- each component could be a separate thread, or even running on a separate processor.

- the whole system could run as one single task, implementing one or more of the five functional components.

- the system could have any structure in between.

The whole system's specifications are ambitious, in the sense that it is difficult to start from an existing software project that implements part of the applications (e.g., ECC signal processing), and extend it to cover all the specifications. The basic difficulty lies in the fact that so many components must cooperate in so many different configurations, without loosing performance. This goal can be reached by *decoupling*, from the start, the functional components via *mediator classes*, Section 3.

3 Mediator classes

The concept of a *mediator class* is the key behind the envisaged flexibility. Instead of letting all components interact with each other directly, they interact with the mediator. Hence, the complexity of providing the code for all possible interactions, is reduced to the code to connect each component to a small number of mediators. These latter interconnections can be made smaller, more uniform and more configurable, *if* one designs the mediators with the whole scope of the generic application in mind. Further complexity reductions emerge by using the *specific structure* that the system exhibits (Section 4).
The presented design is built around the following three key mediator classes:

- The general-purpose mediator classes *Producer-consumer* (Section 3.1) and *events* (Section 3.2). The producer-consumer is for *data exchange*, and the event for *synchronization*.

- The system-specific *Execution Engine* mediator. This one decouples and localizes the *logical interaction and synchornization* between all other functional and infrastructural components.

These mediators are classes in the *object-oriented programming* sense of the word: they *encapsulate data structures that are shared* between the components, and take care of the *synchronized ("race-free") access* to these data. This *localization* and *uniformization* of the IPC interactions between components facilitates understanding of the design and the implementation, and hence also porting, scaling and configuration. This flexibility is further enhanced by the fact that *only two different* general-purpose mediator classes are used, whose implementations are shared by all components.

3.1 Producer-consumer

This mediator takes care of the *data exchange* between two components. The "producer" component generates data, in its own pace; the "consumer" component uses the data, also in its own pace. The mediator makes sure that both components don't have to worry about each other, and even don't have to know each other. The mediator implements the *mechanism* of the data exchange, i.e., the functionality of access to the encapsulated data: shared memory, circular buffer, non-blocking FIFO, etc. So, the producer-consumer mediator *class* consists of the following data and methods:

- *Data.* The data structures exchanged by producer and consumer. In order to reduce data copying, the mediator may offer the service of letting producer and consumer share the same data structure inside the mediator. The mediator takes care of the specified form of access synchronization.

- *Methods.* The classical `read`, `write`, `clear`.

This is the mediator's interface to the producer and consumer components. The mediator, however, also has an interface to the infrastrucure component of the system. This interface is used for *configuration* of the mediator: memory allocation, policy setting, etc.

3.2 Event

This mediator takes care of the *synchronization* between two components. Every component can *register* a function as its *handler* for a particular event; multiple components can register handlers for the same event. When one component "fires" that event, the mediator executes all the registered handlers. One distinguishes between two types of handlers:

- *Listeners.* These handlers are executed immediately after the event fires, and are not interrupted by a possible new firing of the event.

- *Completers.* These handlers are executed at some later time, after all listeners have finished.

The semantics of listeners and completers is exactly like that of Interrupt Service Routines (ISR) and Deferred Service Routines (DSR) for interrupts. The event class has the following data and methods:

- *Data.* Linked lists for registered handlers. Bookkeeping data for event *policy* implementation: What to do with an incoming event, if the processing of the same event is still going on? What about cancelling a previous event? Etc.

- *Methods.* Registration and de-registration of handlers. Event firing. Event handling (listeners and completers).

The event is a very powerful mediator, with which one can build more complex mediators, such as, for example, a *finite state machine*. The state machine is an important synchronization mediator in infrastructure part of the signal acquisition and generation system. Advantages of the event concept are:

- Events can be implemented with very limited RTOS functionality: the minimally required support is a mutex. (For atomic access to the various lists that the event has to manage.)

- Events improve loose coupling between components, especially when one provides the functionality of running only some of the core components.

- Events allow straightforward scaling, not only in size, but also in distribution over a network.

- The event concept is perfectly compatible with hardware and software interrupts: events and interrupts are both asynchronous, and have immediate and deferred handlers (Interrupt Service Routine + Deferred Service Routine for interrupts; listeners and completers for events).

- Events allow different components to be "timed" independently and varying over time. Even to work with a "virtual" time, in order to interactively slow down or speed up the signal processing rate (see the HeartBeat below).

- Events offer a more direct way than classical RTOS primitive (such as static priorities of threads) to implement the synchronization logic of components.
- Events can be implemented in different ways: as method calls on an event object; running its own thread; internally multi-threaded (e.g., one thread for each handler); and even distributed over a network. Hence, the appropriate level of efficiency can be chosen.

4 System architecture

This Section describes in more detail the functional contents of the components. The description uses an *object-oriented* terminology, which hints at the implementation design covered in Section 6. The producer-consumer and event mediators are used everywhere. All components can run at different rates, or act completely event-driven, triggered by the HeartBeat.

4.1 Data flow architecture

Figure 1 sketches the *data flow* in the system, and this shows the *inherent structure* that each Producer-Consumer mediator has only one *writer*. Hence:

- The specification of the data structure classes in the system is nicely localized, to one single mediator at a time.
- There is a *natural sequential order* in the *logic flow* of the system: the Execution Engine must activate the functional components in the following order: (1) Scanner, (2) Observer, (3) Generator, (4) Controller, (5) Actuator, and (6) Reporter. This adds the extra *inherent structure* that all activity in the system is *most optimally executed sequentially*, and that one gains nothing from running different *functional components* in parallel threads. (This does *not* necessarily imply that parallel threads are not needed for, for example, the device drivers, or for the *infrastructural components* of reporting and command interpretation.)
- In multi-threaded versions of the design, *read/write locks* are the most appropriate locking mechanism to be used in the internals of the mediator, because data only flows in one direction.

The *Producer-Consumer* mediators are the basic infrastructure for the *data flow*. Running everything *serialized* in one single thread has an enormous consequence on their efficiency: they don't need any mutual exclusion or context switches! Hence, the implementation of the presented *generic* system gets the efficiency of existing, special-purpose and less flexible systems.

4.2 Logic flow architecture

The *Event* mediators are the basic infrastructure for the *logic flow*. In the case of a single thread implementation, one could also get rid of these *Event* mediators, and replace the logic flow by a single loop in the Execution Engine, much like the classical PLC architecture: during every loop, the functional components are called in sequence, unless their timing indicates that they don't have to be called during that specific loop.
Leaving the Event mediators in makes the system easier to distribute, and the design remains unchanged over whatever scale it is applied. The cost of the Event mediators corresponds roughly to two extra function call indirections (times six components): one for firing the event, and that event in turn calls each of the registered handlers.

5 Plug-in API

The previous Section described the inherent structure in the architecture of the system. That is, all those parts that remain *invariant* between all envisaged applications. The fortunate consequence of such an invariant structure is that the names, numbers, and semantics of all components and mediators are *fixed*. What is not fixed can easily be parameterized, as explained in the following sections.
This Section explains how application programmers use the framework presented in the previous Sections. They have to do the following things:

- To fill in the *functionality* of the functional components.
- To configure the *logic flow*, i.e., the Event mediators.
- To fill in the *data structures* to be managed by the Producer-Consumer mediators.
- To configure the *timing* with which the Execution Engine must execute the various components. (Only the *functional* components, because the application programmers don't "see" the infrastructural components.)
- To configure the protocol and semantics of the messages the Command Interpreter will have to work.

The following subsections give *examples* of what sort of API is needed to fulfill the above-mentioned needs. (The given function calls should not be considered to be complete or optimal.)

5.1 Functionality

```
plugin(component, function);
```

where `component` is one of these: `scanner`, `actuator`, `generator`, `observer`, or `controller`. The `function` is the algorithm to be executed in the component; it should be dynamically linked into the system runtime.

```
plugin_configure(component, data_structure);
```

5.2 Logic flow

This configuration is simple: select which of the available events in the generic system are needed for the application. This corresponds to telling the Execution Engine about which components to activate, and to connect to their mediators. This doesn't need more than a simple

```
plugin(component, status);
```

where `status` can be "on" or "off."

5.3 Data flow

The data flow inputs are clearly localized in each of the Producer-Consumer mediators. So, the application programmers must just fill in the application-specific contents of these generic "containers." A pseudo-code version of these calls is:

```
plugin(mediator, data_structure);
```

`mediator` is (see Fig. 1) one of *inputs*, *estimates*, *setpoints*, or *outputs*. (The syntax above is "C"-like, but, of course, a similar OO method call can be used.) The `data_structure` should be of a format which offers the application programmers flags for the parts they want to be *reported*. These flags can be raised and lowered with function calls such as:

```
plugin_configure(mediator,
                 data_structure_flag,
                 status);
```

5.4 Timing

The `timing` could be as simple as setting desired frequencies; but it could be made more flexible in several ways:

- Allow *don't care* specification of the timing. In which case the Execution Engine will fill this in.
- Allow the specification of a *dependence* on the timing of another component. For example, the Actuator should wait for the Controller to generate a new result before it updates the hardware.

5.5 Command interpreter

The API above consists of multiple configuration commands for the real-time system core. Of course, these commands do not go directly to the functional components, or to the mediators: it's the job of the Command Interpreter to do the bookkeeping of all configuration requests, and check their consistency. At a certain moment, the application program signals the Command Interpreter that all configuration calls given up to now should be applied to the system, *as a whole* and *atomically.* At that moment, the Execution Engine switches to its *reconfiguration state*, which fires events for each of the functional components, asking them to reconfigure themselves according to the specifications collected and checked by the Command Interpreter. (In this reconfiguration state machine, getting completion events back from the components is a useful (but not indispensable) application of the Event mediator principle.

6 Implementation overview

This Section describes how to implement the real-time signal core presented in the previous Sections. Various different implementations can be imagined, corresponding to the various ways in which the mediator classes can be implemented. This document presents the simplest, most efficient approach for a uni-processor system. A more complex approach (and more detailed descriptions) can be found on `http://www.orocos.org/documents/ipc.pdf`.
The Execution Engine runs on a *timer*, whenever any of the components in the whole system requires activation. Its internal algorithm is a state machine, that scans the events to be fired for each of the other components. Each of these components can also be a state machine. The most obvious candidates are the Generator (especially for hybrid signals), the Controller (for switching control algorithms, as wel as for running through its control loop), and the Reporter. This implementation is indeed extremely simple, and hence also efficient: the functionality of all components is executed as *function calls* issue by the Execution Engine, which is itself the only one that needs more support from the operating system. The

efficiency comes from the fact that the design implements its own, application-specific "scheduling," which will perform better than the general-purpose RTOS scheduler, because the latter doesn't profit at all from the inherent structure of the application.
The implementation is straightforwardly supported by the timer functionality of the real-time Linux operating systems. RTAI offers even a more versatile primitive, in the form of its *tasklets*, which can, for non real-time use, be replaced by the Linux tasklets with similar semantics. The C++ support of RTAI is also a plus, because it allows to use the event library developed for the Orocos project.
Implementations for more complex systems than a uniprocessor system just need to replace the event and producer-consumer mediator implementations. For example, the producer and consumer can reside on different computers and have to talk to each other through a network. In this case, the data exchange and event internals of the producer-consumer mediator are easily distributed, without having to change anything to the application clients.

7 Conclusions

Our major experience that underlies the writing of this paper is that advances in real-time applications will not in the first place be driven by more features in the RTOS, but, on the contrary, by investigating each particular application area and looking for (i) the intrinsic *structure* of the application's activities, and (ii) the *minimum* amount of RTOS features needed to implement a generic solution.
Real-time cores for a wide variety of signal processing applications can all be implemented on the basis of a quite straightforward design, as soon as an appropriate decomposition in component modules is made. This paper indentified the following components as necessary and sufficient for building a generic real-time signal processing core: the functional components of the Scanner, Actuator, Generator, Controller, and Observer; and the infrastructural components of Execution Engine, Command Interpreter and Reporter. The connection between all component can be *localized* into a low number of mediator classes (Producer-Consumer, Event, and Execution Engine), which are all well understood, can be implemented efficiently, and, most importantly, which guarantuee a *loose coupling* between all functional and infrastructural components.
However, various applications still have to provide their own *policy* on top of the common and generic real-time core *mechanism* presented in this paper. Also, the problems of user interfacing have not been discussed in this paper.

References

[1] H. Bruyninckx. Open RObot COntrol Software. `http://www.orocos.org/`.

[2] C. A. Culianu and D. J. Christini. RT-Lab—Real-time experiment software for Linux. `http://www.rtlab.org/`.

[3] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building application frameworks: object-oriented foundations of framework design.* Wiley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, Reading, MA, 1995.

[5] B. Koninckx, H. Van Brussel, B. Demeulenaere, J. Swevers, N. Meijerman, F. Peeters, and J. Van Eijk. Closed-loop, fieldbus-based clock synchronization for decentralised motion control systems. In *CIRP 1st International Conference on Agile, Reconfigurable Manufacturing*, 2001.

[6] D. Schleef. Comedi: Linux control and measurement device interface. `http://stm.lbl.gov/comedi/`.

[7] C. Wuollet. MatPLC. `http://mat.sourceforge.net/`.