EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH
CERN - SPS DIVISION

[REDACTED]
CERN LIBRARIES, GENEVA

CERN SPS/89-43 (ACC)

CM-P00061477

SASD and the CERN/SPS Run Time Coordinator

G. Morpurgo

Abstract

Structured Analysis and Structured Design provides us with a handy way of specifying the flow of data between the different modules (functional units) of a system. But the formalism looses its immediacy when the control flow has to be taken into account too. Moreover, due to the lack of appropriate software infrastructure, very often the actual implementation of the system does not reflect the module decoupling and independence so much emphasized at the design stage. In this paper the Run Time Coordinator, a complete software infrastructure to support a real decoupling of the functional units, is described. Special attention is given to the complementarity of our approach and the SASD methodology.

Presented at the International Conference on Accelerator
and Large Experimental Physics Control Systems,
Vancouver, British Columbia, Canada,
October 30 - November 3, 1989.

Geneva, October 1989

SASD and the CERN/SPS Run Time Coordinator

Giulio Morpurgo
CERN/SPS, 1211 GENEVE 23, Switzerland

October 24, 1989

Abstract

Structured Analysis and Structured Design provides us with a handy way of specifying the flow of data between the different modules (functional units) of a system. But the formalism looses its immediacy when the control flow has to be taken into account too. Moreover, due to the lack of appropriate software infrastructure, very often the actual implementation of the system does not reflect the module decoupling and independence so much emphasized at the design stage. In this paper the Run Time Coordinator, a complete software infrastructure to support a real decoupling of the functional units, is described. Special attention is given to the complementarity of our approach and the SASD methodology.

Introduction

The realization of a "software system" is a complex process. The software developer(s) is given as input a natural language description of what the "system" is supposed to do, and eventually he has to produce the actual implementation of the system. The process goes through several stages (analysis, design, implementation, testing, ...). Structured Analysis and Structured Design (SASD) is a methodology aimed to help and guide software developers in the analysis and design stages of a "system". For each of these stages, which can be themselves subdivided in further levels of detail, SASD suggests one or more ways of providing a formalized description of particular aspects of the system (data flow diagrams, state diagrams, etc.). The overall idea is to oblige (and to help) the software developer team to be precise about each particular aspect of the system; nothing will be left undetermined. The prize for the exercise should be a clear and complete description of the system, taking into account all the aspects. This description could also be used as documentation of the way the system works.

SASD, of course, is not a magic formula. "Good" software developers using SASD will still do better than "bad" ones. The same is true for any other methodology. Another point to be emphasized is that SASD is a collection of methods, each of them to be applied to a particular aspect of the analysis and design of the system. In what follows, we want to present a complementary approach, to be used together with an SASD-like analysis in designing a system. A software tool (the Run Time Coordinator) to support the formalism of our approach has been developed by us here at CERN [1] [2]. It helps the application programmer in both the design and the implementation stages of software development. The Run Time Coordinator has been used in several applications for the control systems of the SPS and LEP accelerators.

The SPS/LEP control system working environment

To better understand the reasons and the ideas behind the conception of the Run Time Coordinator, let us describe the environment in which the software to control the SPS and LEP has to run. We have a network of UNIX machines. Programs running on this network are mainly written in C or Fortran. In addition, Unix Scripts are also used.

Some of the computers in the network are interfaced to the hardware of the accelerators; data to be read from the accelerator (orbit measurement, etc.) or to be sent to the hardware (current setting to magnets, etc.) will have to pass through these computers. We call these computers "front-end".

Other computers are "interfaced" to the accelerator operators. Via an easy to use man-machine interface the operators must be able to monitor and control the accelerators. We call these computers "Consoles". In our case the consoles are Apollo workstations, essentially because they offer efficient human interaction (mouse, multiwindowing, colours, graphics).¹

As a typical simple application, let us consider the monitoring of the particle beam in the accelerator. An operator can decide to start the monitoring of the beam; data from the beam monitors connected to the front-end computers must be periodically collected, gathered somewhere, interpreted, and displayed in a graphical format on the console on which the operator is working, until he decides to stop the monitoring.

The problems. As one can see, we deal with a **distributed system**, with parts (components) running on different computers. The implementation of the system implies problems of synchronization, networking and data communication between the different components. A point which is important to underline is that the above described network architecture is a constraint; it will not be up to the software developer to decide for a change in the network, or to use a more appropriate operating system, or a multitasking language like, for example, ADA. These implementation decisions, that in a idealistic view of the process of developing the software system would have to be taken after analyzing the system, are imposed on the software developer. In other words, we have to face the fact that requirements (like "periodically acquire the data") are stated together with implementation details (like "on the front end computer").

The ideas behind RTC. The above mentioned problems (synchronization, data communication, networking, imposed implementation constraints) turn out to be common to a wide set of applications in the accelerator control systems (and, more generally, to any real time distributed application). As a natural consequence of this fact came the idea of providing a software infrastructure to deal with these problems. This infrastructure should lie between the network and some aspects of the operating system on one side, and the application programs on the other side. It should provide facilities to the application programmers in such a way that they do not have to worry at all about implementing solutions to these problems. This infrastructure is called the Run Time Coordinator (RTC). Later we shall show how the RTC is complementary to a part of the SASD methodology. Here we discuss shortly the main features of the

¹ There is also another, older network of mini computers, running an interpretative language called NODAL. Our system enables the application programmers to run programs on this network and to communicate with it.

RTC.

The Run Time Coordinator

Our system is divided in two parts. The first part consists of a "Task Specification Compiler" (TSC). The software developer writes a text file containing a description of his application in a simple language. This file (called the "Task Specification File") is given as input to the compiler, which checks the syntax and produces tables to be used by the second part of the system. This first part is used during the design stage of the application.

The second part of the system is the actual Run Time Coordinator. It consists of a set of "server" processes running on each node on our network. These processes are able to understand the tables produced by the TSC, and to carry on the execution of the application in the way specified by the software developer. This second part deals with all the problems mentioned at the beginning of this section in a way transparent to the software developer.

Let us now have a closer look at the formalism and the functionality of the RTC.

Tasks and processes: the RTC language. In the language of the RTC, an application is called a "task". A task can consist of several "processes", plus the "rules" determining the coordination between these processes. A task could be, for example, the system to monitor the particle beam; the processes belonging to it are the processes to acquire the data, the one to gather all the data, and the process to display it; a rule could be used to represent the requirement

"acquire the data every SPS cycle"

or the requirement

"refresh the display every time new data is gathered"

Note that the RTC supports a distributed environment. Processes belonging to a single task can be specified to run on different computers in the network.

The RTC synchronization primitives. The rules determining the coordination between the execution of the different processes belonging to a task can be basically expressed in the format

execute action when event

The most important actions in the current implementation are

- Execute a process
- Signal a process
- Kill the task
- Start a subtask ²
- Kill a subtask

We distinguish between three classes of events :

² A subtask can be seen as a part of the whole system, itself made of different parts, whose interactions will be analyzed when going to a lower level of detail.

- Events generated by the RTC system itself
 - A process terminates
 - A process is launched
 - A timeout expires
 - An accelerator event ³ occurs
 - A subtask terminates
- Events generated by processes belonging to RTC tasks
 - A process reports a value to RTC ⁴
 - A piece of data is produced
- "External" events, generated by any process running over the network⁵
 - An external event occurs
 - An "external" piece of data is produced

Boolean combination of events can also be used. Finally, sophistication can be added to the rules by the use of **clauses**. It is possible, for example, to specify an automatic repetition of a rule, to set a limit to the number of times the rule has to be repeated, to introduce a delay between the occurrence of the event and the execution of the associated action, to specify a "precondition" to the execution of an action. Processes belonging to a task do not "talk" directly to each other. Rather, they communicate via data stores (shared memory segments, files, etc.) . The RTC takes care of the distribution of these pieces of data if, for example, two processes needing access to the same shared segment are running on two different computers .

The goals of the RTC. To summarize, what we try to obtain by using the RTC is to give the application programmers the possibility of splitting their distributed applications (*tasks*) into simple, functionally decoupled parts (*processes*) which do not contain any explicit reference to other parts of the application. The coordination between the different processes is specified in a text file, called the Task Specification File. In this file the features of the processes belonging to a given task, together with the rules specifying their interactions, are specified in a simple language. Also the data stores needed for the interprocess communication are declared in this file. Modifying the coordination between the different processes of a task will not require any change in the code of the individual processes : this will be done by changing the rules in the Task Specification File. Another positive aspect of our approach is that, because the processes are **functionally decoupled**, it is more likely that they can be reused in other tasks. For instance, we have a general display utility, the DataViewer, which can be used to display data produced by virtually any process.

³ eg. "extraction stops"

⁴ Note that in this, as in all the following events, the process does not talk to any other process of the task. It only talks to the RTC software infrastructure. It will be up to the RTC to decide if the information must have any effect (starting, signalling, etc.) on other processes.

⁵ The difference between this class of events and the previous one is that, while in the former class events are local to the task in question, events of this class will be reported to any task who declared its interest in them. This declaration of interest is also contained in the Task Specification File.

SASD and the RTC

Having said that our processes communicate via data stores, it becomes quite natural, when designing a RTC task, to identify the processes, and the data they access. This is precisely the same kind of modelling that leads to the production of an SASD **data flow diagram**. We have, in fig. 1, rectangles representing data stores, and bubbles, representing processes which "transform" these data (they are called "data transform" in the SASD language). This kind of diagram clearly shows the static aspect of the interaction between the different parts of an application (via the data they share); it does not show the dynamics of the synchronized execution between all these parts. In other words, it does not show the *control flow*. If we now continued our analysis in a SASD fashion, we would use other parts of the SASD methodology to cover all the aspects that still are missing from the data flow diagram, to go into further details. The goal, as we said at the beginning, is to obtain a clear and complete representation of the system we have to build.

The point we want to make here is the following : in our environment, another convenient way of carrying on the analysis of our system is to use the formalism of the Run Time Coordinator. We will therefore express the coordination between the different parts of our system (or, if you prefer, the different processes of our task) via the rules linking actions to events. If necessary, we will go into further details by using a subtask instead of a simple process. All of this is expressed in the Task Specification File in a readable format. Furthermore, the RTC, by taking care of the execution of the task as specified in the Task Specification File, constitutes a powerful software tool supporting our formalism. In practice this means that the application programmer will not have to deal with implementing the synchronization between the different parts of his system, but only with implementing each individual part. Once the synchronization is designed and expressed via the RTC rules, the Run Time Coordinator takes care of it.

An example

As an example, let us consider an hypothetical system to monitor and control the accelerator orbit. The first description of the system comes in natural language :

"The orbit can be changed by modifying the setting values in the *Accelerator Hardware*. This operation is done by the *Set Hardware* process every time a new setting is produced, either by the *Operator*, or by the *Automatic Correction* program. The *Operator* can enable or disable the *Automatic Correction* program. Data measured by the *Orbit Monitors* are periodically read by the *Monitors* programs, then collected and processed by the *Manipulator*. This latest program produces data in a format suitable for the *Display* program, which refresh the display every new cycle, and for the *Automatic Correction* program. This program compares the measured orbit with the target orbit, initially decided by the *Boss*, and computes the required corrections to the Hardware settings via a magic algorithm. Both the *Boss* and the *Operator* can decide to stop the system."

The next step is to formalize this description in a **data flow diagram**, in order to individuate clearly processes and data involved in the system, and their interactions. This is done in fig. 1.

Having done this, it is time to concentrate our attention on the **control flow** in our system. This step will clarify the sequencing between the different processes contained in our data flow diagram. We carry on this operation by using the RTC formalism. Without going into all the details of the RTC language, fully described in [2], the result of the exercise is the following, simplified, Task Specification File.

```
#define kill_process      signal  9
#define takes_over       reports 1
#define gives_in         reports 2
#define says_stop        reports 1
#define wakeup           signal 16

task name: orbitmaster

process name: Boss
data generated: wished_orbit

process name: Operator
host: here
data generated: new_setting

process name: Automatic
data generated: new_setting

process name: Set_Hardware

process name: Manipulator
data generated: processed_data
host: Speedy

process name: Display
host: here

process name: Monitor1
host: front_end1
data generated: data1

process name: Monitor2
host: front_end2
data generated: data2

/* choose the target orbit; start the display program */
exec Boss
exec Display
/* Once the target orbit is chosen, the Operator starts his job */
exec Operator when wished_orbit arrives
/* The Operator can enable or disable the Automatic correction */
repeat uniquexec Automatic when Operator gives_in
```

```

repeat kill_process Automatic when Operator takes_over
/* Set the hardware with new settings every time they are available */
repeat exec Set_Hardware when new_setting arrives
/* Read the orbit from the orbit monitors every cycle. "start_of_cycle"
   is an accelerator event */
repeat exec Monitor1 on start_of_cycle
repeat exec Monitor2 on start_of_cycle
/* Process data when all the monitors have been read */
repeat exec Manipulator when (data1 arrives and data2 arrives)
/* Refresh the Display with new data */
repeat wakeup Display when processed_data arrives
/* Perform a new correction when new data arrives, only if the Automatic
   correction is enabled */
repeat wakeup Automatic when processed_data arrives with Automatic running
/* kill the task (stop everything) if the Operator quits or the Boss is happy ! */
kill orbitmaster when (Operator finished or Boss says_stop)

```

Note that it is possible to specify where a certain process should run. If this process generates data which must be used by another one running in another computer, the RTC will take care of the data transfer in a way transparent to the programmer. In our case data1, produced on front_end1, and data2, produced on front_end2, will be transferred to Speedy, where they are required by the Manipulator process.

In reality, many more properties can be specified in the declaration blocks for tasks and processes. The example reported here is intended to provide an illustration of the RTC system.

Conclusion

We have presented a formalism supported by a software infrastructure (the RTC). The formalism describes the run time behaviour and the interactions between the individual components of a system. The software infrastructure carries on the execution of the system in the way specified by the formalism. This is done without requiring any effort from the application programmer.

The formalism adaptes for application together with an SASD-like analysis leading to a data-flow diagram. These two methods combined can provide a detailed and concise description of the system.

From the implementation point of view, by using the RTC the application programmer will not have to waste time dealing with "system software" problems. His code will also be more portable and more directly aimed at his application, rather than to the solution of these problems. Another advantage of our approach is that each component of the system can be run and tested individually, because it does not contain explicit references to the other components. Finally, if necessary, by changing the Task Specification File it will be straightforward to modify the way the different parts of the application interact, without changing their code.

References

- [1] W. Herr, R. J. Lauckner, G. Morpurgo *The RTC - A Software Support System for the Control of the CERN SPS*. CERN/SPS/89

- [2] G. Morpurgo *The Run Time Coordinator and Task Specification Compiler User Guide*. CERN Lep Controls Note 100

Fig. 1. This is a pure data flow diagram for our example. Circles represent **processes**, open rectangles **data stores**, closed rectangles **hardware**. Note how in this diagram only data flow is represented. Neither the sequencing of the process execution, nor the influence of one process on another (like the *Operator* disabling the *Automatic Correction*) are shown.