DESIGN OF FLEXIBLE AND DEPENDABLE REAL-TIME APPLICATIONS

M. Bizzarri†, P. Bizzarri†, A. Bondavalli∗∗, F. Di Giandomenico† and F. Tarini†

† Scuola Superiore S. Anna, Via Carducci 40, Pisa, Italy
∗∗ CNR-ENEA, Via Santa Maria 36, Pisa, Italy
† IBI-CNR, Via Santa Maria 46, Pisa, Italy
‡ Università di Pisa, Dip. Informatica, Corso Italia 40, Pisa, Italy

Abstract

The design of safety-critical applications must include redundancies aimed at Fault Tolerance purposes. An adaptable use of such redundancies can optimise resource utilisation; in case of faults, it can preserve fundamental services and allow a graceful degradation of the system. A programming notation, named FERT, has been recently introduced for the design of adaptable applications. With this notation, the designer of a safety-critical Real-Time application can specify alternative fault tolerance policies for each component of the application and obtain an adaptable run-time behaviour. This paper is meant as a contribution to this notation and to its possible implementation. Namely, it deals with the specification of a communication semantics (including failure semantics) and with execution support problems such as the definition of value of a computation and adaptive planning at run-time. Some related issues are also addressed as future work.

Keywords Real-Time Applications; Fault Tolerance; Flexibility; Design Description Language.

1. INTRODUCTION

Many Real-Time applications control safety-critical systems with high dependability requirements. To fulfil them, Fault Prevention techniques are necessary, but do not allow to exclude the occurrence of faults at run-time; therefore, also Fault Tolerance (FT, in the following) policies must be employed, for example Recovery Block (Randell, 1976) or N-Modular Redundancy (Avizienis, Chen, 1977). The necessary redundancy can be either statically sized according to the worst case, or exploited by an adaptable behaviour of the application. The latter solution, in case of overload or lack of resources due to failures, allows a graceful degradation of the system: e.g. safety-critical tasks could be supported as long as possible, while non critical ones could receive fewer resources or even be omitted. Generally speaking, when more than one application compete for system resources, the run-time support should perform allocation and management of the resources. For this purpose, specific needs of each application must be adequately specified. This specification could also allow an integration of FT policies in the design and implementation of the application and in the validation of its Real-Time requirements. A possibility in specifying the flexibility in resources demand of each application is to express each of the interacting components of the applications by means of several alternative computations; each of them implements a different FT policy and thus includes a different level of redundancy and requires a different set of resources. At run time, only one of these alternatives will be selected and executed, according to resources availability. The programming in-the-large language used to specify these alternatives should also allow to associate to an alternative a variety of information including a value (i.e. the system benefit coming from its execution) and a cost (i.e. resources necessary for its execution). The run-time support, which plans the execution of applications, may choose the appropriate alternative for each component, based on the associated costs and values and resources availability, trying to optimise the global system benefit.

A programming notation has been proposed in (Bondavalli, Stankovic, Strigini 1993) and named FERT (“Fault tolerant Entity for Real Time”). That proposal does not completely define semantics and is not detailed about the implementation. This paper aims at a contribution to the FERT notation; namely, it investigates the specification of communication semantics (including failure semantics) and addresses some of the problems related to its implementation, namely...
the definition of the value of a computation and problems of dynamic planning.

The remaining of this paper is organized as follows. Section 2 outlines the FERT notation as proposed in the above cited paper. Section 3 is devoted to specifying the communication semantics; both functional and fault semantics are considered. Section 4 deals with implementation issues, mainly about dynamic guarantees for real-time constraints; the problem of stating the value of a computation is also addressed. Section 5 contains conclusions and indications for future work.

2. THE FERT NOTATION

The FERT notation introduces an intermediate level in the design of the application, between the in-the-large and the in-the-small programming; at run-time, an application is composed of a set of functional modules called FERT.

A FERT includes three parts, each described below: 1) an interface towards the other FERTs; 2) a set of Application Modules (AM in the following); 3) a control part which describes how to handle the execution of the AMs. An interface is simply composed of a set of input and output ports. The AMs are modules written in a language in-the-small, each one with its own worst case execution time and its input and output ports. The control part controls the dynamic execution of the AMs in a FERT. In the control part the application designer can define an execution graph for the AMs through four primitives: POSSIBLE, EXEC, UNUSED and OUTPUT. The POSSIBLE primitive defines an alternative for the execution of the FERT, intended as a set of AMs with precedence constraints, a deadline for completing its execution and a value associated to it. With more than one POSSIBLE, the FERT designer can implement alternative computations ("strategies") for the same FERT, with different values and costs.

The remaining three primitives state how and under which conditions the execution of a single strategy must occur. The EXEC primitive specifies the inputs needed for the execution of each AM; such inputs may come from the output port of other AMs, from input ports of the FERT or may be given as constants. The OUTPUT primitive sends results out of the FERT, through the FERT interface. Last, the UNUSED primitive states that some AMs are no longer necessary to complete the strategy; such primitive is invoked after a run-time control on some condition of the output ports. For example, in a Recovery Block (Randell, 1976) the next alternative is not executed, if the results of previous one pass the acceptance test.

Externally, the FERT behaves also as a fault containment unit: the FERT resolves and masks all the faults admitted under the failure hypothesis made. Therefore the application designer can define the FERTs and their interfaces as parts of a functional design, without any explicit concern of faults and fault propagation between FERTs. Moreover for each mode of functioning of the system the FERT's belonging to an application can be divided in fundamental and non fundamental ones. The fundamental FERTs execute the computations that in the current mode of functioning are critical to the integrity of the system or for safety reasons if the run-time support accepts to execute a given application, then it must guarantee that at least a default strategy (defined by the designer) will be executed within its deadline, of course under given fault and load hypotheses. The arrival time to the system of the fundamental FERTs of an application can be periodic (equally spaced, with a period characteristic for the FERT) or sporadic (arbitrarily spaced, but always with an interval longer than a fixed minimum, characteristic of the FERT). Non-fundamental FERTs can also be aperiodic (arrival time arbitrary spaced without a minimum).

The original proposal of this notation does not specify a semantics for the activation of the AMs and for communications among them. In this paper, the following simple assumption is made: the execution of an AM is triggered by the availability of its input data, possibly coming as output data of other AMs. As in a dataflow approach, AMs communicate to each other only at the beginning of the execution (to get input data) or at its end (to give results of their computation).

3. COMMUNICATION SEMANTICS

The semantics of communication must be specified both in case of correct behaviour of the whole system (functional semantics) and with respect to the faults which are allowed to propagate to the upper level (fault semantics). This semantics must cover the communications between AMs belonging to the same FERT (infra-FERT communication) and belonging to different FERTs (inter-FERT communication).

In principle, a unique semantics could be suitable for both communications. However, the infra-FERT communication semantics should aim at facilitating the specification of different FT policies, both general-purpose and application-oriented, and these policies may need different failure semantics, with different implementation costs. Moreover, the inter-FERT communication is to be applied to a purely functional design of the application. Therefore, the inter-FERT communication should have a no-failure semantics, that is, no fault has to be allowed to propagate to the upper level, because no FT provision are normally taken there. Obviously, the reliability required to the inter-FERT communications may vary, depending on the application constraints. These considerations dissuade from unifying the two semantics.

3.1 Functional semantics

Both infra-FERT and inter-FERT communications are aimed at exchanging two classes of information: events and information about the current status of some real-time entity, e.g. the value of physical quantum
Infra-FERT communication semantics. The semantics of the communication among AMs within the same FERT must take into account that, after being started, the execution of an AM continues till completion without any communication with other AMs. Only at the end of its execution, an AM can make available its results to other AMs. It may be unknown which AMs will actually use these results, because different possible sets of AMs can be actually executed during the execution of a strategy. More, when a message is sent, the receivers in general are not executing. This means that the sender of a message cannot synchronize with the receivers, since it may not even know who they are. A model for this behaviour is the following: the AMs communicate using objects, called ports, where the sender can write its messages, and from where the receivers (possibly more than one) can read them. The port is thought as belonging to the FERT to which both the sender and the receivers belong. The ports inside a FERT constitute the environment where each AM is executed. This environment is renewed each time the FERT is executed. An AM is executed (and described) as a procedure: first, it binds its formal parameters to the messages present in the port; then, it executes its own body, according to the values of the messages through read operations, and through write operation it can produce new messages which are put on its output ports. It will be necessary to define different objects (with different data structure and different semantics for the operations) according to different characteristics of the communication, namely overridable information or not and different failure semantics as it will be discussed later.

Figure 1 is an example of specification of an overridable port, in a Pascal-like language with objects. The example is meant to express the port functionality over a distributed system, and it is therefore divided in two different definitions: the SOPort (Single Overridable Port), implementing the low level functionality; and the DOPort, (Distributed Overridable Port) representing the abstract object to which the AMs may adhere. Both ports have four methods for according to the messages. The init method initializes the port and is invoked at every execution of the FERT. The bind method binds the name of an AM to a message (in this case, the last message arrived). From there on, through the read method, the AM can adhere only to that message, even if new messages are written on that port. The write method writes a new message. The replicated keyword means that the data structure has to be replicated all over the nodes where the FERT is distributed. This set of objects is referred to as p in the example. The local keyword allows to refer the local replica of the object, while the forall keyword allows to specify a command to be executed on all the replicas. In the example, pc is used to identify the replicas of p.

type SOPort = object(T)
var buff: array[1..MAX_MSG] of T;
b: 0..MAX_MSG;
begin
  binding: array [AM_NAME] of 0..MAX_MSG;
procedure write(var v: T);
begin
  bi := bi + 1; buff[bi] := v end;
procedure bind(n: AM_NAME);
begin
  if bi > 0 then binding[n] := bi
end;
function read(n: AM_NAME): T;
begin
  if binding[n] > 0 then
  read := buff[binding[n]]
  else read := NO_VALUE_ERROR
end;
procedure init;
begin
  foreach a of AM_NAME do binding[a] := 0;
  bi := 0
end;
end;

type DOPort = object(T)
var p: replicated SOPort ;
procedure write(var v: T);
begin
  forall pc of p do pc.write(v) end;
procedure bind(n: AM_NAME);
begin
  forall pc of p do pc.bind(n) end;
function read(n: AM_NAME): T;
begin
  read := local(p).read(n) end;
procedure init ;
begin
  forall pc of p do pc.init end;
end;

Fig 1. An example of overridable port

Inter-FERT communication semantics. While ports for infra-FERT communication are part of the FERT environment, different FERTs have disjoint local environments. This suggests the message passing model for the inter-FERT communication. Decisions must be taken regarding when the messages produced are sent to another FERT, when the sent messages are received and when the receiving AMs can accede the message contents. The following choices are part of this proposal.

- A communication channel (which may be one-to-many) is represented by a set of ports, one for each receiver. In the inter-FERT communication, receivers are defined in a static way and may change only with a change of the system mode. Therefore, they are assumed to be fixed.
at the time the sender FERT is executed.

- A FERT sends a message to each receiver of the communication channel: this means that each receiver will have its copy of the message; in this case the communication does not use shared objects.

- A FERT sends messages at any time during its execution: when the strategy issues an inter-FERT communication, sending a message to another FERT, it must have reached enough confidence in the message sent. This confidence must have been reached despite the system, due to unexpected faults, cannot guarantee that a FERT will continue its execution after sending a message. For example, a message could cause a tap to be open, intending that a successive message issued by the same FERT will close it. If the FERT happens to crash in the mean time, that tap could remain open forever.

- A FERT can receive a message at any time (after it has been sent); the transmission delay may be subject to application requirements that must be satisfied by the support.

- The AMs belonging to a FERT have access only to those messages arrived before its current execution starts. This means that the FERT input ports should be frozen when execution starts. It remains to be decided what is the exact meaning of “the start of an execution of a FERT”. Of course the freezing of ports is a quite restrictive constraint that could lead to limitations of the expressive power of the FERT approach.

Figure 2 shows an example of infra-FERT and inter-FERT communications. The arrows in the infra-FERT communication are meant to show the accesses of the receivers to the communication ports (through the `read` method). Instead, the arrows in the inter-FERT communication represent the message passing from the (environment of the) sender to the (environment of the) receiver (triggered by the `OUTPUT` primitive).

3.2 Failure semantics The failure semantics of a system (in this case a communication sub-system) deals with the definition of the set of the different behaviours that the system shows, should it fail to provide its service. Examples may be:

i) none: the system is always required to behave correctly and any deviation from the expected behaviour is a violation of this failure semantics;

ii) all: every failure does not violate the failure semantics;

iii) the communication subsystem may lose some messages but those messages that are delivered are delivered correctly; as long as the system either behaves correctly or omits to deliver messages the semantics is not violated.

A system with an explicitly defined failure semantics is more suitable to be seen as a component of another enclosing system: the design of the enclosing system has only to cope with the permitted failures of its components.

Intra-FERT communication. Both a higher reliability and a more restrictive failure semantics have a cost in terms of system resources. The choice of these levels, as well as the balance between them given a certain amount of available system resources, depends on the application requirements. These can be satisfied basically in two possible ways: dealing with communication errors and/or providing more reliable ports.

Processing of communication errors not masked by the underlying system is a proper job of the strategy. For example, the FERT in figure 3 implements a recovery block. At each activation, the FERT processes
the first message available in its input port IN. The primary module P is executed and the acceptance test
AT checks its result, P.O. If the test is not successful, the backup copy B is executed, and AT tests its output B.O. If one of the tests is successful, the result is sent out of the FERT (an OUTPUT is performed). If a communication failure occurs when P “reads” from
IN, P produces an erroneous value, detected by AT, if the AT itself is correct. This communication failure may be recovered within the strategy, provided the communications performed by B and the execution of B itself are successful. As shown in this example, AMs do not process communication errors; at most, they could detect errors and possibly decide whether to continue or to terminate execution.

To provide more reliable ports, the basic ports available as the basic communication mechanisms supported by the underlying system may be used to build up ports with the same functional behaviour, but with more reliability and/or with a more restrictive failure semantics. Note that the AM designer should not build up his own ports; rather, he ought to have a library, from which he can choose ports with a reliability level and a failure semantics suitable for his needs.

Figure 4 presents an example of a built-up port, named TOPort (Triplicated Overridable Port); the TO-
Port is meant to provide tolerance to one arbitrary value error, or to detect two or three non coincident value errors. The TOPort uses three DOPort in order to detect and possibly to mask such errors. Since the DOPort, as previously defined, are replicated on all the nodes of the FERT, a copy of the three DOPort is present on each node of the FERT. For the sake of brevity, the algorithm used to perform adjudication (for example, voting) is not specified.

Inter-FERT communication. FERTs are entities implementing, possibly in a fault tolerant way, a function of an application. Fault tolerance is provided inside FERTs and failures in inter-FERT communications can’t be processed by an higher level. Then, the only useful characterisation of inter-FERT communication is the reliability. Of course, the unreliability of the inter-FERT communication sub-system will contribute to the unreliability of the whole application. Still there can be different reliability requirements for different applications or parts of an application. Thus, it seems reasonable to structure the communication sub-system so that different reliability levels are provided (different costs); the application designer will have to select the most appropriate communication service. Again this could be accomplished by providing in the communication layer different types of ports with different reliability provisions.

4. THE RUN-TIME SUPPORT

Required Features. The run-time support must manage the system resources in order to achieve the following properties:

- execution of the fundamental FERTs within their deadlines. Due to their importance for the integrity of the application, a given minimum level of performance must be statically guaranteed. This behaviour of the system can be proven under given faults conditions; however these conditions are only stochastically predictable, possibly with high probability;
- a system-level optimisation of computational resources assigned to each FERT. The assignment should optimise the value accrued by the selection of the strategies; to this purpose the assignment policy should include an arrival model for non-fundamental FERTs, so that resources can be reserved for high-value FERTs not yet arrived;
- a graceful degradation of the total accrued value in presence of overloads (due to bursts of arrivals or to faults). Most important FERTs should be supported, while less important ones receive fewer resources or are completely removed.

A Model of the System. To discuss the implementation, a model is proposed hereby, made of three different entities: a set of computation nodes, a population of FERTs and a Planner.

The computation nodes are connected to each other (e.g. by a local network or by a shared memory) and constitute a distributed system. Each of them can perform tasks with given start time and deadline.
At run-time the population of FERTs generates instances of FERTs. Resources for fundamental FERTs are statically allocated, at least those resources required for their default strategies. Several algorithms are available for this static allocation: for example, fixed priority in conjunction with server mechanisms (Sprunt, Sha, Leloczky, 1989); fixed priority with static (Leloczky, Ramos-Thuel, 1992) or dynamic (Audsley, Davis, Burns, 1994) slack calculation; dynamic priority with server mechanisms (Buttazzo, Spuri, 1994). The Planner examines non-fundamental FERTs and plans their execution, using computational resources not allocated to fundamental FERTs, plus resources gained during their execution; namely, resources are gained by unused AMs and by executions shorter than the worst case execution time. The Planner examines also the fundamental FERTs, trying to guarantee the execution of a strategy better than the default one. For each FERT, the Planner selects one strategy to be executed, according to the associated value, deadline and resource requirements. The AMs of these selected strategies are allocated to computational nodes, where their execution is locally scheduled. Note that planning is a system-level activity aimed at fulfilling precedence constraints, allocation requirements and value optimisation for groups of tasks, while scheduling is a node-level function, responsible of the execution ordering of tasks, so that each task scheduled can meet its deadline. In a multiprocessor architecture like Spring (Stankovic, Niehaus, Ramamrithan, 1993), a Planner can also perform the scheduling activity, so that computation nodes are only concerned with dispatching activity. A Planner can be implemented either in a centralised way, e.g. with a node dedicated to planning activities or in a distributed one, where a component (hardware or software) on each node cooperates with other components to perform planning activity. The components of a distributed Planner must co-ordinate to each other through a protocol, so that the local workloads can be taken into account while planning the execution of an entire strategy.

At a first glance, the Planner could have a complete view of the load of each node, simply memorising the load it assigns to each node, plus the static allocations to fundamental FERTs. However, this is a pessimistic view of node load, based on the hypothesis that each AM of the selected strategy is executed, and lasts for its worst case execution time. This pessimistic view would force pessimistic choices of the Planner, which could reject a FERT even if enough resources are available for its execution. An interaction between the nodes and the Planner is then required, so that the Planner can get feedback information about gained time and can adjust its pessimistic view. A centralised Planner may try to minimize communications with (possibly remote) nodes, by keeping memory of assigned loads and static allocations. Instead, a distributed Planner is likely to interact more strictly with nodes; as it has a component on each node, it can directly interact with the scheduling queue of the node.

Existing Hard Real Time kernels. Presently, a kernel offering all the characteristics needed to implement the FERT notation is not available. However, a first analysis shows that some kernels exist (at least for centralised systems) with useful features such as: management of periodic, sporadic and aperiodic processes and of their deadlines; static guarantees for fundamental tasks; system calls for knowing whether it is possible to fulfil timing constraints of a given process; bounded delay for message transfer. These kernels include: HARTIK (Buttazzo, 1993), DRTEE (Audsley, Burns, Richardson, Tindell, Wellings, 1991) and Spring (Stankovic, Niehaus, Ramamrithan, 1993). To implement the FERT notation on an existing real-time kernel, one or more of the following actions seem to be feasible: to implement a layer of software that improves the features of the kernel used; to sub-set the notation according to its features; to impose restrictions to the FERT designer while defining strategies. For example, a FERT notation including only imprecise computations (Liu, Lin, Shih, Yu, 1991) and non-adaptive FT strategies (i.e. always employing a fixed number of resources, like NMR) can be implemented on a target kernel without any high level planning feature. In fact, the unused statement has not to be supported and each FERT is composed by only one strategy, which can be either an imprecise computation or a fixed set of AMs statically allocated to each node. On the other hand, it could be possible to develop on some existing RT-kernels a high level co-ordination protocol like that presented in (Di Natale, Stankovic, 1994). This protocol should co-ordinate local schedulers of a distributed system in order to dynamically select and guarantee a strategy for incoming FERTs; this protocol could be realised at the process level on each of these kernels, so to extend the available features without modifying the kernel itself.

Value of strategies. The FERT notation as proposed in (Bondavalli, Stankovic, Strigini 1993) associates a single numerical value to each strategy of a FERT. This definition can be not enough flexible and expressive. To perform a good planning activity, the Planner should have the following information about the system benefit related to a strategy and to the FERT to which the strategy belongs:

- the FERT importance. The importance of fundamental FERTs is not comparable with the importance of not fundamental ones;
- the functional value of the FERT, considered as a top-level component of the application;
- the completion probability of the strategy. A strategy has a probability of completion as a measure of its tolerance to software and hardware faults; in different application environments a Planner has to make trade-offs among redundancy and number of FERTs accepted for execution;
- the result precision of the strategy. There are many computational models like imprecise computations (Liu, Lin, Shih, Yu, 1991) and unbound computations (Audsley, Burns, 1994).
In fundamental FERTs, the choice whether to execute the default strategy or a more expensive one can be based on completion probability and/or result precision.

The high complexity of a Planner using all this information suggests to investigate some approximations. The information about a strategy should be synthesised (partially or completely) and the quality of the resulting Planning should be compared to that of an ideal Planner which uses complete information.

5. CONCLUSIONS AND FUTURE WORK

In an environment oriented to envisage realisation issues of the FERT notation, this paper i) proposes a specification of both functional and failure semantics for communication among application components; ii) outlines the model of a system for implementing the notation and addresses execution support problems such as the definition of value of a computation and run-time planning. Each of these issues is open for future work. About the communication semantics, some questions still have to be answered: one of them is when the input ports should be “frozen”, during the execution of a FERT; the simple solution here proposed does not specify the exact meaning of “start of an execution of a FERT”. Simple definitions of this expression can be devised, but further investigations are needed to verify how they suit to needs of target applications. Another open question is the definition of the library of “built-up” ports for the infra-FERT communications. Mostly, it must cover all the most important application requirements without overloading the designer and the development system with too many different alternatives.

Regarding the value, a trade-off must be investigated, between detailed information (FERT importance, functional value, completion probability and result precision) and the algorithmic and/or computational complexity of the Planner. Algorithmic complexity can lower the robustness of the planner (and requires more memory at run time); computational complexity overheads planning activity and reduces the system performance. A case study is felt to be necessary, concerning a meaningful application such as robotics mission control or railway traffic control. This would allow: i) to better understand whether the FERT notation is suitable to represent the real application, and possibly lead to a more detailed proposal of the notation itself; ii) a deeper knowledge of the features needed for its implementation.

AKNOWLEDGEMENTS

This work has been partially supported by the CHECK ESPRIT BRA 6362 PDCS 2 (Predicatably Dependable Computing Systems 2) and by the EC-US Exploratory Collaborative Activity “EC-US041-AFERT”.

REFERENCES


