Adaptive Software Fault Tolerance Policies with Dynamic Real-Time Guarantees

Paolo Bizzarri*, Andrea Bondavalli*, Edgar Nett**, Hermann Streich**, Fabio Tarini*

* CNUCE-CNR, I-56126 Pisa, Italy
** Research Division of Responsive Systems, German National Center for Computer Science (GMD)

D-53754 St. Augustin, Schloß Birlinghoven
-draft version-

Abstract

Real-time applications with high dependability requirements demand for fault tolerance strategies. While for small systems with static behaviour policies based on worst case execution times can be used, this is not true for more complex systems, in which worst case execution times are partially unknown or differ drastically from their average execution time. In such cases often only a minimum of quality can be achieved. This paper proposes to combine fault-tolerant policies described by the FERT (Fault-tolerant Entity for Real Time) notation with the dynamic scheduling scheme TPS (TaskPair-Scheduling). TPS alleviates FERT’s precondition of completely known WCETs and provides a flexible implementation base to enable an easy mapping of FERT strategies to a runtime system. In a first step, a significant subset of FERT is investigated, which implies: the Recovery Block Scheme, N-Version Programming, and Imprecise Computations. TPS is utilised to guarantee different levels of quality, tailored to the application and the required level of fault tolerance, while guaranteeing, that a common deadline is met.

1 Introduction

For Real-Time applications requiring high dependability, Fault Avoidance techniques, although necessary, do not allow to exclude the occurrence of faults at run-time. Then, Fault Tolerance (FT, in the following) policies must be employed. The necessary redundancy can be either statically sized on the worst case, or exploited by an adaptable behaviour of the application. In case of overload or lack of resources due to failures, adaptable use of redundancy can allow 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.

To reach an adaptive use of redundancy, the application designer and the run-time support must co-operate in such a way that the designer specifies needs and flexibility of the application, while the run-time support takes them into account in managing system resources. This specification could also allow to integrate Fault Tolerance policies in the design and implementation of the application, and in the validation of its Real-Time requirements. Special notations have been proposed to this purpose, both as programming-in-the-large notation [Nie91] and as notations meant at an intermediate level [BSS95] between programming-in-the-large and programming-in-the-small. These notations have to be general, powerful and expressive, in order to give the designer the maximum freedom. With the FERT notation, fault tolerance strategies can be defined for complex real-time systems and tailored to the existing environment.

One of the problems of providing fault tolerance in real-time applications is, that fault tolerance strategies are often very time-(resource-) intensive and that worst case execution times are often difficult to estimate. Imprecise estimations also add to more resource consumption which may lead in the end to unfeasible schedules. On the other hand, the effective resource consumption may often be much smaller than the estimated one, which means that the possibility for a successful computation before a given deadline is large. The problem for the run-time system is to guarantee, that given deadline is not violated, even if a computation does not succeed within its timing bounds.

The TaskPair-Scheduling (TPS) scheme [Str95] matches this requirement. It provides a
method for flexible and predictable execution of tasks with hard timing constraints (no deadline violation allowed) in presence of “unsuccessful” computations as described above, by implicitly scheduling exceptional reactions if a task does not terminate within its time bound.

In this paper, ideas about the implementation of FERT-strategies on TPS are described. On the one hand TPS allows use of specifying average execution times while guaranteeing that deadlines are not violated, and on the other hand - if all worst case execution times are known (and usable) - it provides the full guarantee needed for the “original“ FERT notation.

As a first step only a subset of FERT is considered. It consists of fault tolerance strategies which cover a large range of applications: the Recovery Block Scheme [AnL81], N-Version Programming [ChA78], and Imprecise Computations [Liu91]. In this paper the mapping of the Recovery Block Scheme to TPS is described as an example.

The rest of this paper is organised as follows. Section 2 outlines the FERT notation; section 3 outlines the TPS dynamic scheduling scheme; section 4 introduces restrictions in the use of FERT notation and addresses some issues about usability of TPS to realise FERTs; section 5 specifies a reasonably complete set of FT techniques and proposes for some of them a possible implementation in terms of TPS.

2 The FERT notation

The FERT notation (Fault-tolerant Entity for Real Time), proposed in [Bondavalli, Stankovic, Strigini 1995], allows the application designer both to define adaptable behaviour of the application and to specify different FT policies for each application component. This notation introduces an intermediate level in the design of the application, between the programming in-the-large and the programming in-the-small; at run-time, an application is composed of a set of functional modules, also called FERTs.

A FERT includes three parts, each described below (see fig. 1):

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, 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. Lastly, 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 FERTs 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).

In short, at run-time only one of the strategies of a FERT is chosen for execution, according to its „cost“ in terms of resources needed and its „value“ in terms of benefits gained by its execution. When accepting to execute a FERT, i.e., one of its strategies, the system is expected to guarantee deadline and internal precedence constraints of this strategy. The system must also guarantee that it will accept to execute possible safety-critical FERTs arriving with periodical rate or with a sporadic rate, although with an inter-arrival time not less than a given a minimum. Non critical FERTs can contend resources to each other, possibly utilising resources left because of unused redundancies and/or because of executions shorter than the specified worst case execution time (WCET).

3 The Task-Pair Scheduling scheme

The Task-Pair Scheduling scheme (TPS) is a central component of the DIRECT approach aimed to dynamically guarantee to each application component accepted for execution, a minimum level of computation, and to execute the remaining part based on a best effort approach. To this purpose, the component is structured in a Task Pair (TP), made of a SoftTask (ST) and a HardTask (HT). A deadline is specified for the TP. The ST executes the normal computation, with a „best effort“ approach. The HT is executed in case of violation of the pair deadline; its WCET is guaranteed if the pair is accepted for execution.

The application programmer specifies the two parts (the SoftTask and HardTask, according to the TPS notation) while he’s writing the code of the process itself.

```
TaskPair TP = (SoftTask,HardTask);
```

with the intention:

```
if( guarantee (TP,deadline)){
  try
    <SoftTask>
  except <SoftTask>
  end
}
else // no guarantee
```

The TaskPair is described by a TaskPair descriptor, including a task descriptor for each task providing all necessary information about the TaskPair to the on-line scheduler, e.g. WCET (if possible) or a function to compute the WCET, earliest start-time, dependencies etc..

The basic idea was to execute the SoftTask only if all required resources of the HardTask can be reserved by the scheduler. This guarantees that either the SoftTask or the HardTask will succeed the operation before the deadline. Only the WCET of the HardTask must be known. It is obvious, that such a construction makes sense, if the WCET of the HardTask can be computed, and is small in comparison to the WCET of the SoftTask.
The smaller and simpler a HardTask is, the larger is the chance to guarantee it. The question is, what can or should effectively be done by a HardTask?

We decide between three main functions:

1. Undoing, what the SoftTask has done so far, which can be used for implementation of atomic behaviour of TaskPairs.

2. Completion of what has been tried by the SoftTask, e.g. for kinds of imprecise computation, the HardTask can utilise, what has been done so far by the SoftTask to compute a usable result.

3. Support of „basic functionality“, which is applicable to graceful degradation schemes. The HardTask performs a minimum quality of service, that is used if the desired quality cannot be achieved.

If the HardTask is empty (but exists), TPS guarantees the termination of the TaskPair before its deadline. No application relevant computation is performed, the only purpose of the HardTask is to avoid a deadline violation and to avoid the disturbance of other tasks.

In either case, predictability of the timing behaviour of the TaskPair is guaranteed.

A Simple Guarantee Algorithm For TaskPairs

The simplest way to implement TaskPairs is, to dispatch SoftTasks with a simple dispatch strategy, preemptive and with low priority, while executing HardTasks as late as possible but with maximum priority. This means that HardTasks are planned in a „time triggered“ way and that they are non-preemptive. A simple dispatch strategy can be e.g. EDF or least laxity first in combination with guaranteeing a minimum of execution time for SoftTasks. In this paper, guarantees for SoftTasks are not considered and round robin is assumed as dispatch strategy, although it is not optimal for optimising work of SoftTasks.

The following picture shows two independent TaskPairs TP1 and TP2 running concurrently.

```c
void TP1(deadline)
{
    if( guarantee(TP1,deadline) ) {
        try ST1();
        except HT1();
    } else ...; // no guarantee
}

void TP2(deadline)
{
    if( guarantee(TP2,deadline) ) {
        try ST2();
        except HT2();
    } else ...; // no guarantee
}
```

Figure 2: Simple strategy for scheduling TaskPairs

The scheduler places the HardTasks (HT1 and HT2) as late as possible with respect to their deadlines\(^1\) (DL) by reserving processor time on a time axis. In this example the deadline of TP2 is earlier than HT1’s deadline. The SoftTasks (ST1 and ST2) start running. ST1 and ST2 may be preemptive or not and scheduled in an arbitrary way. If ST2 does not terminate before start time (ST) of HT2, HT2 starts execution. If ST2 succeeds, i.e. ST2 terminates before ST(HT2), HT2 is removed from the scheduler’s plan. TaskPair T1 behaves analogously.

TaskPairs may be nested, i.e. within SoftTasks and within HardTasks further, TaskPairs may be entered. The scheduler knows about the dependency, i.e. an inner TaskPair has always to be terminated before the outer one.

Currently distributed scheduling is under development. Communicating local schedulers guarantee TaskPairs with a common deadline executing on different nodes.

\(^1\)For simplicity we do not consider time for run-time system activities like guaranteeing tasks, switching of threads etc.. All such activities must be bounded (see also [Stan89]).
4 Issues on implementing FERTs on a TPS system

Due to the expressive power and generality of the FERT notation, this paper considers only the mapping of a subset to TPS; furthermore, practical direction for investigation seems to introduce slinth modifications both on FERTs and on TPS.

About FERTs, a different kind of guarantee for strategies and some restriction to the usability of the notation can be introduced, still allowing a satisfactory degree of flexibility.

One of the basic assumptions of FERTs is that at run-time system resources are allocated to the selected strategy before its execution, such that its deadline is guaranteed; while implementing FERTs on TPS, in order to maintain this assumption each strategy ought to be mapped on the HardTask of a pair, leaving empty the corresponding SoftTask. This method cannot be very effective; a different assumption is proposed hereby, trying to exploit at best the flexibility of TPS: while accepting a FERT for execution, instead of committing just one of its strategies, the system just guarantees the execution of the „minimal” strategy; later, during execution, it attempts to complete a „better” one if possible, on a best-effort basis. To take full advantage of this approach, it seems appropriate to build FERTs by incremental strategies. A strategy A is incremental with respect to a strategy B, if A is made by adding components and functionalities over B. For example, A could include an accurate but fragile algorithm and, as a second chance, a robust but less accurate one; B could be made just of the latter algorithm. In this case, the two strategies could be easily mapped on a Task Pair, where strategy B is always guaranteed by putting the second algorithm as the HardTask, still strategy A could be executed depending on the system run-time conditions, by putting the first algorithm as the SoftTask.

About TPS, it must be noted that redundancies introduced for fault tolerance purposes cannot be managed based only on deadlines: if a task completes within its deadline, but produces a wrong result, then a recovery task must be executed, typically as the HardTask of the pair. Moreover, a task can complete within its deadline still requiring execution of further tasks in order to decide whether or not its results are acceptable; this is the case, for example, of N-Version Programming schemes. This implies that a SoftTask must be allowed to indicate, while terminating, whether or not its HardTask counterpart has to be executed, even in absence of deadline violation.

Mapping in TPS FERTs made of incremental strategies.

Consider first the special case that the FERT is made by only two strategies, S1 and S2; as they are incremental, one of them (say S1) includes the other one, S2. The most natural way to map them is to put S2 as the HardTask, and the rest of S1 as the SoftTask. Unused primitives possibly used in the strategy need not to be mapped, they are implicit in the functioning of the TPS. Validation functions (e.g. Acceptance Test or Adjudication Function) must be included in the SoftTask; conditional statements used in conjunction with validation functions can be rendered just encapsulating them in the validation itself; the resulting object performs the validation test and, in case of success, it outputs the result; else, it avoids termination of the SoftTask, thus causing the execution of the HardTask.

In the general case that FERT is made by more than two strategies, S1 . . .Sn, ordered as above, the mapping can utilise TPS nesting: it puts Sn in the HardTask and the rest of S1 in the SoftTask; the latter, its turn, will be made of a pair, where the Hard component maps S(n-1) minus Sn and the Soft component maps the rest of S1, that is S1 minus S(n-1); this last, at its turn, will be made of a pair, etc.

To investigate concrete problems of implementing FERTs on a TPS system, the following set of fault tolerance policies can be taken into account, as representative ones: Recovery Block, N-Version Programming, and Imprecise Computations; these three FT policies seem to be sufficient for a
programmer to specify a good variety of solutions covering almost every real-life scenarios. Recovery Blocks and Imprecise Computations have a similar dependency structure like TPS, so their mapping can take advantage of that. For N-Version Programming, a sequential execution of the Version can be mapped; it has to be decided which version(s) to execute in the SoftTask and which in the HardTask. If parallelism is required (e.g. to execute different versions on different processors at the same time or quasi-parallelism on one processor but with different threads of execution), a TaskPair have to be spawned, i.e. at least asynchronous activities has to be started, which have to inherit the initiating TaskPairs deadline; more precisely, it has to inherit the start time of the HardTask of the initiating TaskPair. In such cases, the HardTask of the TaskPair could be used for comparing the results of the N-Versions.

5 Implementation of some FT techniques by means of TPS.

In the following, two examples are proposed of implementation of FERTs on TaskPairs.

The first one is based on a Recovery Block scheme, the second one on a N-Version scheme.

In both examples, a normal TPS notation is used, enriched with the primitive execHardTask, which means that a SoftTask, while completing execution within its deadline, requires the execution of its Hard counterpart.

**Recovery Block**

The FERT is supposed to include following incremental strategies:

- $S_1$ a Recovery Block with two versions;
- $S_2$ the execution of a single variant with test on the output;
- $S_3$ an exception handler.

The strategy $S_1$ has the following code:

```
begin {Strategy S1}
exec(V1(Fert.In), AT(V1.out));
if AT.out=succes then begin
    output (V1.out)
    unused(RS,V2,AT,RS,EH)
    end
else begin
    exec(RS,V2(Fert.In,
```

AT(V2.out))
if AT.out=succes
then begin
    output (V2.out)
    unused(RS, EH)
    end
else exec(RS, EH)
end {Strategy S1}

In the code, the following symbols have been used:

- $V_i$ is the i-th variant of the RB;
- AT is the module implementing the acceptance test;
- RS is the component restoring the state after a failed execution of $V_i$;
- EH is the FERT exception handler.

The strategy $S_2$ has the following code:

```
begin {Strategy S2}
exec (V2(Fert.In), AT(V2.out))
if AT.out=succes then begin
    output (V2.out)
    unused(RS, EH)
    end
else exec(RS, EH)
end {Strategy S2}
```

Lastly, the strategy $S_3$ is the following:

```
begin {Strategy S3}
exec(RS, EH)
end {Strategy S3}
```

According to the idea proposed in the previous section, these strategies can be implemented by means of a nested TPS, like the following one:

```
TPFERT = { HTFERT, STFERT }  
HTFERT = {RS, EH} 
STFERT = TRB = { HTRB, STRB}  
HTRB = {RS, V2, AT, 
    if AT.out=succes then output(V2.out) 
} 
STRB = {V1, AT, 
    if AT.out=succes then output(V1.out) 
}
```

As said above, the „unused“ is ignored. Up to this point, it is not clear which entity is in charge to execute the „if“ part. As said above, this part can be encapsulated in a new Acceptance Test, say EAT1, which when
necessary also requires, while terminating, the execution of the Hard counterpart; it can behave as follows:

\[
\text{begin \{EAT}_i\}
\begin{align*}
\text{AT}_i&(V_i.out) \\
\text{if} \ &\text{AT.out=success} \\
\text{then} &\text{output (V}_i\text{.out)} \\
\text{else} &\text{execHardTask}
\end{align*}
\text{end \{EAT}_i\}}
\]

Using such an EAT, the mapping becomes as follows:

\[
TP_{FERT} = \{HT_{FERT}, ST_{FERT}\}
\]

\[
HT_{FERT} = \{RS, EH\}
\]

\[
ST_{FERT} = TP_{RB} = \{HT_{RB}, ST_{RB}\}
\]

\[
HT_{RB} = \{RS, V_2, EAT_2\}
\]

\[
ST_{RB} = \{V_1, EAT_1\}
\]

Apart from RS, this implementation is expected to execute, in normal conditions, \(V_1\) and then, if necessary, \(V_2\) and then, if necessary, \(EH\); \(V_2\) is necessary in case of deadline-missing of \(V_1\) (e.g. due to a temporal failure), or in case that \(V_1\) gives a wrong result. It executes only \(V_2\) (and then, if necessary, \(EH\)) in case of scarce resources; \(EH\) is necessary in case that both \(V_1\) and \(V_2\) miss their deadline or give wrong results. It executes only \(EH\) in case of extremely scarce resources.

\[\textbf{N-Version Programming}\]

The FERT is supposed to include following incremental strategies:

- **S1**: a N-Version scheme, with three versions;
- **S2**: a N-Version scheme, with two versions;
- **S3**: the execution of a single version plus an acceptance test;
- **S4**: an exception handler.

In the code of the strategies the following symbols have been used:

- \(V_i\) is the i-th version of the NVP;
- \(EH\) is the FERT exception handler;
- Voter is a component which behaves according to the number of results on which it executes. It behaves as a voter if there are three results; it behaves as a compare test if it is called on two results; it behaves as an acceptance test, if it is called on a single result; last, it does nothing if it is called on no result at all.

The strategy **S1** has the following code:

\[
\text{begin \{Strategy S}_1\}
\begin{align*}
\text{exec}&((V_1(Fert.In)||V_2(Fert.In)||V_3(Fert.In)), \\
\text{Voter}&(V_1.out, V_2.out, V_3.out)); \\
\text{if} &\text{Voter.out=success} \\
\text{then} &\begin{align*}
\text{output} &\text{(V}_3\text{.out)} \\
\text{unused} &\text{(EH)}
\end{align*} \\
\text{end} \\
\text{else} &\text{exec(EH)}
\end{align*}
\text{end \{Strategy S}_1\}}
\]

The strategy **S2** has the following code:

\[
\text{begin \{Strategy S}_2\}
\begin{align*}
\text{exec}&((V_2(Fert.In)||V_3(Fert.In)), \\
\text{Voter}&(\text{\(-, V_2.out, V_3.out\))}; \\
\text{if} &\text{Voter.out=success} \\
\text{then} &\begin{align*}
\text{output} &\text{(V}_3\text{.out)} \\
\text{unused} &\text{(EH)}
\end{align*} \\
\text{end} \\
\text{else} &\text{exec(EH)}
\end{align*}
\text{end \{Strategy S}_2\}}
\]

The strategy **S3** has the following code:

\[
\text{begin \{Strategy S}_3\}
\begin{align*}
\text{exec}&((V_3(Fert.In)), \\
\text{Voter}&(\text{\(-, \text{\(-, V_3.out\))}); \\
\text{if} &\text{Voter.out=success} \\
\text{then} &\begin{align*}
\text{output} &\text{(V}_3\text{.out)} \\
\text{unused} &\text{(EH)}
\end{align*} \\
\text{end} \\
\text{else} &\text{exec(EH)}
\end{align*}
\text{end \{Strategy S}_3\}}
\]

Lastly, the strategy **S4** has the following code:

\[
\text{begin \{Strategy S}_4\}
\begin{align*}
\text{exec} &\text{(EH)}
\end{align*}
\text{end \{Strategy S}_4\}}
\]

These strategies can be implemented by means of a nested TP, like the following one:

\[
TP_{FERT} = \{HT_{FERT}, ST_{FERT}\}
\]

\[
HT_{FERT} = \{\text{Voter, if Voter.out = success} \\
\text{then output (V}_3\text{.out) else EH}\}
\]

\[
ST_{FERT} = \{\text{HT}_{NVP}, ST_{NVP}\}
\]
HT_{NVP} = \{V_3, \text{execHardTask}\}
ST_{NVP} = \{HT_{NVP1}, ST_{NVP1}\}
HT_{NVP1} = \{V_2, \text{execHardTask}\}
ST_{NVP1} = \{V_1, \text{execHardTask}\}

Again, the "unused" is ignored and the execHardTask means require the execution of Hard counterpart even if completing execution within deadline.

In normal conditions, this implementation is expected to execute V_1, V_2 and V_3 plus the Voter plus, if necessary, EH. It executes only V_2, V_3 and the Voter (and then, if necessary, EH) in case of scarce resources. It executes only V_3 and the Voter (and again, if necessary, EH) in case of scarcer resources. In case of extremely scarce resources, only execution of EH is committed.

\section{Conclusion}

In this paper it is outlined how fault tolerance strategies for real-time applications can be implemented. Strategies denoted by the FERT notation are mapped to the TaskPair-Scheduling scheme.

While for small systems with static behaviour fault tolerance policies based on worst case execution times can be used, this is not true for more complex systems, in which worst case execution times are partially unknown or differ drastically from their average execution time. Since FERTs originally depend on guaranteed execution times in such cases often guarantees cannot be given, although there may exist a good chance to succeed. Usage of the TPS scheme alleviates the precondition for executing FERT's strategies by mapping them to TaskPairs, which consists of a precisely (guaranteed) scheduled HardTask and a SoftTask, scheduled by best effort.

A possible mapping to TaskPairs was proposed for the Recovery Block and for the N-Version scheme. Similar mappings are under study for other strategies, e.g imprecise computation.

\section{References}


