Most of the systems intended to support software fault tolerance in application programs (abbrev. AP) adopt either of two complementary strategies:

* a set of low level facilities (such as for instance, replicated objects, reliable communication protocols, and reliable files manipulation) are provided to the AP programmer, who has to coordinate their use within his/her software;

* or some particular language construct is provided, to take checkpoints and perform rollbacks.

We propose an approach which combines the two above, by describing a programming environment which exploits the separation of concerns between the design of an AP and that of making it fault tolerant. In [1], we proposed to adopt a layered software architecture, where at least three main levels can be identified: Kernel, Application and Recovery layers (as shown in Figure 1). Actions needed to enforce fault tolerance are defined separately from the AP, and are implemented by the so called Recovery Meta Program (abbrev. RMP), which executes in the Recovery layer. The AP provides the "bricks" for the fault tolerant constructs, such as, for instance, validation tests and redundant code, whereas the RMP defines the semantics of the constructs, joining the bricks together. The switch between AP and RMP is performed by the kernel, which also provides the primitives needed by the RMP to implement such fault tolerance actions.

In our programming environment, a generalized RMP is a collection of library modules, each one implementing a specific strategy. These modules can be automatically adapted to support a selected combination of fault tolerance techniques in APs.

The programming environment supporting the development and use of a RMP should be able to satisfy requirements from two types of programmers:

- the developer of an AP, wishing to use existing RMP procedures to implement some strategy within his/her software, as much as possible in automated way.

Consider this latter type of user, who is expecting the programming environment to behave as follows:

i) An AP is submitted to a translator, together with some simple means of identification of the 'bricks' for fault tolerance. Thus, according to the selected language and interaction possibility, the source language may be extended with a few keywords, to identify them. Otherwise, a dialogue may be used to select procedures, labels within them, and the like, leaving the AP unchanged.

ii) Step i) is iterated for each module composing the AP, and then, a tool is invoked for AP building. Such a tool must also be able to retrieve the relevant RMP portions from the library and to adequately configure the run-time support for the given AP/RMP combination.

iii) Finally, the AP and its comprehensive run-time support are ready for execution.

This structure is shown in figure 2.

Let us consider an application process containing the following Recovery Block:

\[
\text{RB: } \begin{align*}
\text{ensure acceptance test} & \quad \text{by 1-st (primary) alternate} \\
& \quad \text{else by 2-nd alternate} \\
& \quad \ldots \\
& \quad \text{else by n-th alternate} \\
& \quad \text{end;}
\end{align*}
\]

\text{exit} \ldots
where RBi is the label (and identifier) of the Recovery Block, the acceptance test is a boolean expression, an alternate is some code portion (block, with one entry and one exit) and exit is a label.

Now, we can state more precisely how to achieve the separation of concerns between the AP and the RMP. The bricks provided by the AP are the different code portions and labels, while the RMP works on these components giving them the desired shape. Therefore, other components of the programming environment are needed to specify and to properly combine such bricks.

As for program translation, we shall then assume that it consists on three basic steps:
1. at a high level of abstraction, the relevant code portions must be identified (e.g. by preprocessing of source text);
2. the AP is translated into object code by an ordinary language compiler;
3. the identification of 'bricks' has to be expressed at a lower level of detail, in terms of code addresses and the like, for the subsequent steps (postprocessing).

The output of the translator on Example 1 is shown in figure 3, where Kcall is the name of a run-time system procedure. The purpose of Kcall is to perform the context switch between AP and RMP.

```
RBx.enter: Kcall(RBx);
RBx.1: 1-st alternate;
   Kcall(RBx);
RBx.2: 2-nd alternate;
   Kcall(RBx);
...
RBx.n: n-th alternate;
   Kcall(RBx);
RBx.AT: acceptance test;
   Kcall(RBx);
RBx.exit: ...
```

Figure 3: Translator output

The translator also builds a table containing the initial values for the objects managed by the RMP.

For the sake of completeness, let us briefly outline what should be provided as well in the programming environment for the development of RMP procedures.

Assuming that a new version of some RMP procedure has been written, such source code has to be translated and kept in the RMP library. Hence, at least a RMP translator and a librarian (to insert, update, remove the RMP procedures) will be needed. The latter shall maintain, together with RMP code, whatever information is needed for its use in any AP, namely, information for the AP translator, (to describe the building blocks to be extracted from the AP) and information for the RMP builder (for kernel interfacing).

A sketch of the above is given in Figure 5

```
RMP Compiler
   \↓
   RMP object code information to
   (to library) AP translator
```

Figure 5: Development of new RMP procedures

This approach also exploits the reusability of fault tolerant software, since RMP procedures are developed once and for all, and they may be later employed in different applications. This program separation allows an easier system testing, increasing its reliability. Moreover, the RMP may be written in a high level language, facilitating in this way its testing and/or verification.

References