STRUCTURED SOFTWARE FAULT-TOLERANCE WITH BSM

Andrea Bondavalli*, Luca Simoncini**

* CNUCE-CNR, Via S. Maria, 36 - 56126 Pisa - Italy
** Dipartimento di Ingegneria dell'Informazione, Universita' di Pisa, Pisa, Italy

Abstract

This paper presents a structured way of inserting software redundancy in programs and to describe the solutions provided for programming software fault-tolerance techniques. It is based on a data-flow like programming paradigm, which is more suitable, in our opinion, and as shown in [5] , to implement fault-tolerant systems, with high levels of flexibility and performability, than conventional imperative programming paradigms. The proposed computational model, BSM [4], describes an application in a set of atomic modules, mainly functional, which: 1) maintain the visibility of the semantic of the application, in order to take full advantage of the possibility offered by the use of assertions and predicates for early error detection, and 2) maintain a close correlation between the logical structure of the application and the physical support, to take full advantage of replication as a mechanism of redundancy. The set of modules is executed asynchronously, with a firing rule similar to that of data-flow model; the modules are atomic and do not interact or communicate with other modules during execution, but release data only at their termination. The close correlation between the semantic of the application and the module structuring allows also to scale the needed redundancy since it can be properly driven by the semantic of the application itself.

1. Introduction

Structured software fault-tolerance includes those techniques where redundancy, both for detecting and correcting errors, is applied to the individual blocks of application software, based on the semantics of each block, and with the goal of masking errors internal to the block; each technique is characterized by a peculiar way of structuring the interactions among redundant parts, which gives a common syntax and a way of managing the complexity added by fault-tolerance. These techniques are primarily recovery blocks, N-version programming and N-self checking programming, with a number of intermediate or combined techniques.

The design of a system needs the use of a system control and application language which is able to express and manage concurrency and parallelism and is suited for the implementation of structured software fault-tolerant techniques. There are two main motivations for this statement: the first is the reduction of the overhead associated to those techniques, like recovery blocks, which use time redundancy for implementing software fault-tolerance, and the second is that the structure of N-version and N-self-checking programs requires the ability to manage concurrency, possible distribution and parallelism implied by the use of variants which are to be executed in parallel. Such system control and application programming language must have the possibility of managing non-determinism and of describing history sensitive computations. Non-determinism is necessary whenever several processes may compete for the acquisition and the use of resources in a non predictable way, while history sensitivity is needed to allow resource management. The other important features required to such a language are the possibility of structuring programs with a very high degree of modularity, the atomicity of the modules for obtaining early error detection and confinement and the capacity of expressing structured constructs for software fault tolerance. BSM is a system control and application programming language, which satisfies the requirements previously mentioned, based on a data-flow like programming paradigm, which is more suitable, in our opinion, to implement fault-tolerant systems, with high levels of flexibility and performability, than conventional imperative programming paradigms.

2. BSM and its Language

The grammar of BSM, expressed using Backus-Naur formalism is the following:

This work has been partially supported by the CEC in the framework of the Esprit BRA Project 3092 PDCS, the Italian CNR in the framework of Progetto Speciale "Programming Environments and Architectures for Distributed System Development" and partially by MURST Project 40% "Architetture Convenzionali e non Convenzionali per Sistemi Distribuiti".

0-8186-2755-7/92 $03.00 © 1992 IEEE

< program > ::=
"PROG" <prog_id> "::=" < module > { < module > } "END".
< module > ::= "MOD" < mod_id > "::=" < mod >.
< prog_id > ::= < identifier >.
< mod_id > ::= < identifier >.
< mod > ::= < simplemod > | < ndmod >.
< simplemod > ::= < act > < block > < term >.
< act > ::= "IN" "(" < couple$_1$ > { < couple$_1$ > } ")".
< term > ::= "OUT" "(" [< couple$_2$ > { < couple$_2$ > }] ")".
< couple$_1$ > ::= "(" < mod_id > "," < var_id > ": " < type > ")".
< couple$_2$ > ::= "(" < mod_id > "," < var_id >")".
< var_id > ::= < identifier >.
< type > ::= < identifier >.
< ndmod > ::= [< head >] "[" < gc > { "[]" < gc > } "]" [< tail >].
< head > ::= < act >.
< gc > ::= < g > " -->" < block > < term >.
< g > ::= < act > ";" < pr > ";" < bp > | < act > ";" < pr > |
< act >.| < act > ";" < bp > | < pr > ";" < bp > | < bp > |
< pr >
< tail > ::= < block > < term >.
< pr > ::= "PR" "=" < integer >.
< bp > ::= <Boolean expression >.

Any BSM application is structured as a set of non cooperating modules < mod > so that the constraint of synchronizing partners on communication is relaxed. No communication is allowed inside the modules and information is allowed to flow to other modules only on termination of a module. A < program >, prefixed by PROG and ended by END, is composed of a set of these modules, which may be either < simplemod >, simple modules, or < ndmod >, non-deterministic modules, which will be considered later. < simplemod > is composed of:

- an <act >, list IN of < couple1 >, each labeled by (< mod_id >, < var_id >: < type >), containing the identifiers of a module and of a typed variable, e.g. IN { (M1, A: integer) (M2, B: real) };
- a < block > which is described in any language (for example a set of C or Pascal commands or a LISP function etc.);
- a < term >, list OUT of < couple$_2$ >, each, if present, labeled by (< mod_id >, < var_id >), containing the identifiers of a module and a variable identifier, e.g. OUT { (M_1, A) (M_2, B) }.

The < block > cannot refer anything external to the module; that is, no interaction with other modules can be present inside it. IN is the primitive for both activation and input and is the first instruction of each module. The list of < couple$_1$ > provides the comprehensive set of all information the module receives (imports) from the external modules. OUT is the primitive for both termination and output; it is the last instruction of all modules. Its list of < couple$_2$ > contains the set of variables whose values must be exported with the name of the destination module associated to each variable; this list may be void if the module does not export results outside. When the OUT is executed, the values of the variables which appear in the list are sent to the proper destination modules and then the module terminates. IN, on the contrary, does not determine any activity in the module in which is executed. The list of < couple$_1$ > is a set of local typed variables together with the names of the source modules which send the values to be associated to these variables. The effect of the execution of IN is to initialize the status of a module and to activate it. If a module M declares a set of source modules in its IN list, on termination of the last of its source modules, all the variables which M has declared in its IN list will have an assigned value and therefore M can be made ready for execution.

A < program > is structured in a set of functional modules, whose dimension can be varied almost at will (it may consist of a single instruction or a piece of code which does not have external interactions). Interactions are all at module's interfaces, and perform not only communication by sending and receiving values, but also perform synchronization, activation and termination. In this sense our computations is structured as a data-flow graph of modules, which are activated by a firing rule similar to the firing rule of data-flow computations: a module is activated as soon as it receives all data necessary to its function. A module operates on input data and terminates by releasing output values to other modules. In this sense modules can be considered atomic. Nesting among modules is not allowed in this structure: all modules are at the same level of hierarchy, and visibility among modules is not constrained. Limitation to visibility can be introduced only if we want to take into account protection among modules. The set of modules is static: they are defined in a static way and have static rights, which always hold. Therefore most of the control can be performed at compile time by verifying the rights that each module has. The simplest control is the matching of names between partner modules: if MOD M_1 has in its OUT list the couple (M_2, var_1), then MOD M_2 must have in its IN list the couple (M_1, var_1: $type_1$), and the types of the value exported by M_1 and the input variable of M_2 must match. Each module has visibility towards all the other modules which are explicitly named in its IN and OUT lists.

The other module defined in BSM is < ndmod >, which describes the non-deterministic construct; it is used to allow for the description of cyclic computations. Allowing the data-flow graph to contain cycles permits several execution of modules. In this way information related to the status of the modules are preserved, and history sensitive computations are allowed, but all is modeled as usual inputs to modules and not as an internal status of them. < ndmod > has the following structure:

- it may have a < head >, list IN of < couple$_1$ >;
- it must contain a non empty list of < gc >, guarded commands;

- it may have a < tail >, composed of a < block > and a < term >, list OUT of < $couple_2$ >.

< ndmod > is composed of a non empty list of guarded commands. A guard < g > in our model is structured as follows:

- it may have an < act >, list IN of < $couple_1$ >;
- it may have an associated priority < pr >, which is an integer;
- it may have a Boolean predicate < bp > on variables which belong to the IN list.

Guards may be verified, suspended or failed; a guard is verified if the IN list is complete and the values of the variables on which the predicate is expressed, verify it. A guard is suspended if the IN list is not complete and, if the predicate exists, the values of the variables on which it is expressed, do not make it false. A guard is failed if the predicate is false, aside from the presence of all input values. < tail > is present only if all guards may fail; that is, if a predicate is present in each guard and if a set of input values exist which may make the predicates false. < head >, if it exists, is an IN list. The semantics of < ndmod > is as follows: < ndmod >, as any other module, is always waiting for the input values which shall be provided by the predecessor modules on their termination. As soon as one of the guards is verified, the < block > associated to the guard < g > is executed and if more than one guard is verified the proper < block > is chosen in a non-deterministic way. The execution of < block > is performed as in any other module whose IN list is the union of the < head > list and the list associated to the selected guard. The input values which are consumed by the execution of a guarded command are all the values contained in both the < head > list and the IN list, corresponding to the chosen guard. The used values are consumed in the sense that they are no longer usable by other guards which may contain some of the couples of the executed guarded command. When all guards fail, the execution is that of a module whose IN list is the one specified in < head >, and whose < block > is specified in TAIL, which must exist. In this case, all values in the < head > and IN lists of < ndmod > are consumed, restoring the initial status of < ndmod >.

BSM model does not exactly correspond to the data-flow model, since the latter implies that each module is executed as soon as all input values have been provided. This is not true in < ndmod >, which is executed as soon as one of the input guards is verified or when all the guards are failed. The consequence is that < ndmod > can be executed even if not all the inputs are provided, since a guard fails when the input values received make the associated predicate false. Therefore the interpreter must be able to provide a non strict evaluation of the predicates and to execute modules whose input values are not all present. Actually BSM is an extension of the data-flow model, and, due to this extension, it has been possible to provide non-determinism on input by the use of guards with IN lists, beside the usual non-determinism on Boolean guards. Another feature that we want to point out is the wide range in both the dimension and in the complexity of < block > which can be a single instruction or a complete non interacting program written in any language. Due to the close correlation between the BSM modules and the processing units on which these modules are allocated and run, it is possible to give a graphical representation, using boxes and arrows, of a BSM program, in which each box represent a < mod >, either simple or non-deterministic, and arrows between two boxes represent the typed communication channels corresponding to the existence in the source module of < $couple_2$ > in its OUT list, which matches the corresponding < $couple_1$ > in the IN list of the destination module. The orientation of the arrows allows a simple and direct view of the flow of data in the BSM program.

3. Redundancy Mechanisms in BSM

In this section, we consider the mechanisms needed for the insertion of structured redundancy based on the use of i) predicates and checkers and ii) replication and adjudication, and how they can be expressed in BSM. Even if no assumption is made on the hardware architecture for supporting the BSM language, some redundancy at this level is required so that both the hardware support to communication and the BSM interpreter can be considered reliable.

3.1. Predicates and checkers

This mechanism consists in evaluating an assertion in predefined points in the program and if the assertion is evaluated false the program state is recognized as incorrect and inconsistent, and an handler of the abnormal situation is invoked. If the assertion is evaluated true, the normal flow of execution is followed. This mechanism is easily expressed in BSM by inserting between the modules, other modules which implement the assertion in the desired form. Let us consider assertions which verify both the acceptability of an input set of values and/or the consistency of a set of output results; in this case it is usual to speak of predicates. A predicate is implemented by a MOD Chk that is inserted in the flow between two modules such that the values produced by the first one must be validated before the firing of the second.

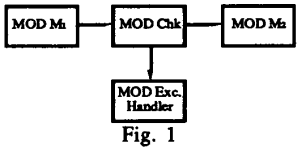

Fig. 1

Fig. 1 describes the structuring of the part of a program after the insertion between MOD M_1 and MOD M_2 of a module MOD Chk performing an assertion on the results produced by MOD M_1. This kind of structuring is very general. The structure of MOD Chk and the modifications to the original structuring are always the same; what is different from case to case is the boolean predicate, which we call Pred (var_1...var_m), which depends on the semantics of the program and the MOD Exc. Handler, whose strategy may vary, depending on the errors detected and the policy it uses.

The structure of MOD Chk is the following:

MOD Chk ::= IN {(P_1, var_1: $type_1$) ... (P_m, var_m: $type_m$)}
[Pred(var_1...var_m) --> OUT{(S_1, var_1) ... (S_k, var_k)}
[] Not (Pred (var_1...var_m)) -->
OUT{(Exc.Handler, var_1) ... (Exc.Handler, var_k) }]

P_1 ... P_m are the predecessor modules, S_1 ... S_k are the successor modules and var_1 ... var_m are the data which are checked. If the goal is to verify the acceptability of an input set of values then S_1 ... S_k are the same module; if the goal is to verify the consistency of a set of output results then P_1 ... P_m are the same module.

We give now an example of an acceptance test on the values which MOD Z receives as input from MOD P and MOD Q. Modules P, Q and Z have the following structure:

MOD P ::= IN{(.., ..)} $block_2$ OUT{(Z, x)}
MOD Q ::= IN{(.., ..)} $block_3$ OUT{(Z, y)}
MOD Z ::= IN{(P, x: $type_x$) (Q, y: $type_y$)} $block_1$
OUT{(.., ..)}

Fig. 2 shows the structure of the program section before the introduction of the acceptance test:

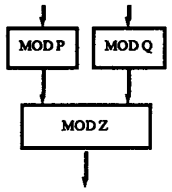

Fig. 2

If we wants to verify the acceptability of values x and y before the activation of MOD Z by means of the predicates $Pred_x$ and $Pred_y$ the structure, shown in Fig. 3, is obtained and the modules now are the following:

MOD Chk ::= IN {(P, x: $type_x$) (Q, y: $type_y$)}
[$Pred_x$ (x) AND $Pred_y$ (y) --> OUT{(Z, x) (Z, y)}
[] Not ($Pred_x$ (x) AND $Pred_y$ (y)) -->
OUT{(Exc.Handler, x) (Exc.Handler, y)}]

MOD P ::= IN{(.., ..)} $block_2$ OUT{(Chk, x)}
MOD Q ::= IN{(.., ..)} $block_3$ OUT{(Chk, y)}
MOD Z ::= IN{(Chk, x: $type_x$) (Chk, y: $type_y$)} $block_1$
OUT{(.., ..)}

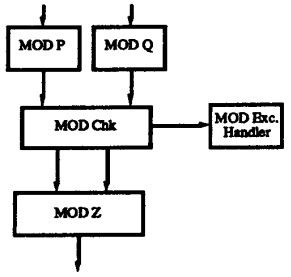

Fig. 3

This way of structuring determines the use of two different modules, MOD Z for computing the results produced by MOD P and MOD Q, and a separate MOD Chk for the evaluation of the assertion, thus allowing a physical separation, if required, of their functions.

3.2. Replication and adjudication

Replication requires the parallel execution of i (i > 2) replicas of the same computation; if the replicas different, they will be called variants. When they terminate, the results are validated by special modules, called adjudicators, which may detect errors occurred in the replicas, and may propagate the results. It is possible both to detect an error and to mask it by ignoring the values adjudged as incorrect by the adjudicator. This mechanism may be used for masking hardware failures if the replicas implement exactly the same function in the same way, or may be used to mask software failures in case of variants, as in the well known technique of software diversity.

3.2.1. Replication of < simplemod >: We start considering the replication of a < simplemod > MOD M and we will consider the replication of an < ndmod > in section 3.2.3. For the replication of a < simplemod >, we must use a Distributor MOD Distr which distributes the inputs to the several replicas MOD M_i of MOD M. Since MOD M_i are < simplemod >, and due to the atomicity of MOD Distr we have not to take into account problems related to the ordering of the inputs, the transformation for structuring the replication of MOD M is:

1) the replicas MOD M_1, MOD M_2, ..., MOD M_i of MOD M are created; each replica has in its IN list the same values, originally in the IN list of MOD M, for which the sender is the Distributor MOD Distr; the OUT list contains the same variables of the original OUT list of MOD M and all the values are sent to MOD Adjud;

2) the OUT list of each predecessor module is changed so that MOD Distr becomes the receiver of the values previously received by MOD M;
3) MOD Distr is created, its IN list is the same as MOD M had before and its OUT list contains for each value received a couple towards each replica MOD M_j, j = 1..i;
4) MOD Adjud is created; it is a < ndmod >; its < head >IN list contains all the couples related to the output values from all the replicas. The decision is performed by an adjudication predicate Adj.Pred. on the values to be checked and, if the test is successful, an adjudication function Adj. Fnct. chooses the list var-$list_{adj}$ of adjudicated values, and the OUT list of MOD Adjud contains all the couples originally in the OUT list of MOD M, with the list var-$list_{adj}$ of adjudicated values to be propagated to the successor MOD S. Alternatively, the OUT list must activate a MOD Exc. H. for the handling of the erroneous situation.
5) The IN list of MOD S are changed by inserting MOD Adjud as sender for the couples in which the replicated MOD M was the sender.

The structure of MOD Distr, MOD M_j and of MOD Adjud is the following:

MOD Distr ::= IN {(P, var-list)}
OUT{(M_1, var-$list_1$) (M_2, var-$list_2$) ... (M_i , var-$list_i$)}

MOD M_j::= IN {(Distr, var-$list_j$)} $block_j$
OUT{(Adjud, var-$list_j$) }

MOD Adjud ::= IN {(M_1, var-$list_1$) (M_2, var-$list_2$) ...
(M_i , var-$list_i$)}
[Adj.Pred. (var-$list_1$, var-$list_2$, ..., var-$list_i$) -->
Adj.Fnct. < choose var-$list_{adj}$ > OUT{(S, var-$list_{adj}$)}
[] Not (Adj.Pred. (var-$list_1$, var-$list_2$, .., var-$list_i$)) -->
OUT{(E.H., var-$list_1$) (E.H., var-$list_2$) ...
(E.H., var-$list_i$)}]

It is straightforward to implement different adjudication functions in BSM. The exact voting. MOD Adjud which implements exact voting is described as:

MOD Adjud ::= IN {(M_1, var-$list_1$) (M_2, var-$list_2$) ...
(M_i , var-$list_i$)}
[< ∃ Majority (var-$list_k$), k = 1..i > -->
< choose var-$list_{adj}$ > OUT{(S, var-$list_{adj}$)}
[]< NOT ∃ Majority (var-$list_k$), k = 1..i > -->
OUT{(E.H., var-$list_1$) (E.H., var-$list_2$) ...
(E.H., var-$list_i$)}]

Implementations which uses a different Adj.Pred. and Adj.Fnct. are simply obtained, by properly coding the adjudication predicate and function in the MOD Adjud

3.2.2 Replication of < ndmod >: In presence of a non-deterministic behaviour it is generally not possible to guarantee that the several replicas are in consistent states during the computation, i.e., have exactly the same behaviour. The problem of replication of non-deterministic modules preventing state divergence is very well known and lot of work exists in the literature. To avoid this divergency a sufficient condition is to impose to the various replicas always to make the same choice. Analyzing the concept of non-determinism it is possible to recognize two causes of non-deterministic behaviour. The first form of non-determinism depends on the internal state of the module or on some hidden condition. A module which shows this form of non-determinism makes choices about its future behaviour independently of any external condition and in a way that is not known from outside. The second form is due to the communication between a module and its environment. It is not predictable which messages will be received by a module, in which order and at what times, and, depending on these external inputs, the behaviour of the module may be different. This form of non-deterministic behaviour is usually called communication non-determinism. If we consider different replicas of the same module that run concurrently on a system for dependability purposes, our goal will be to limit divergencies among them only in consequence of faults and not of different correct choices that can be done in intermediate points of the computations. The only way to avoid divergency among different replicas of a module which behaves according to the first form of non-determinism is to provide the supporting machine with the necessary provision for assuring that the same choice will be made by all the replicas, and it is necessary to depend on it. In the case of the replication of a module which exhibits non-determinism on input, we can simply assure that the input messages are received by all the replicas in the same order for all of them. A general solution that follows this approach is in [8]. As an example of this problem, let us consider the replication of the following MOD C:

MOD C::= [IN {(A, var_1: $type_1$)} --> $block_1$ OUT{(.., ..)}
[] IN {(B, var_2: $type_2$)} --> $block_2$ OUT{(.., ..)}]

Let us suppose to replicate MOD C as in Fig. 4, which shows the communication among MOD A, MOD B and MOD C_1, MOD C_2 and MOD C_3:

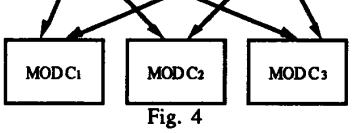

Fig. 4

MOD C_1, MOD C_2 and MOD C_3 are the three replicas of MOD C. It could be guarantied that if the same input messages are presented in the same order, the same guard will be selected by each of them, but, due to errors in the communication or simply to different communication delays, the three copies may receive different input messages or the same input messages in different order. For instance, due to communication delays, MOD C_1 may have as input the message sent by MOD A, MOD C_2 the message sent by MOD B and MOD C_3 may have both messages.

In BSM it is possible to identify a set of modules which guarantees that all the replicas of a given module will receive the same input tokens in the same order, thus assuring that the same guard will be selected for execution by all of them. First of all, with regard to the possibility of loosing messages, we must consider the atomicity of the BSM modules that must be guaranteed by the supporting machine. Since a module is atomic, its correct termination assures that all the output values have been correctly delivered and it is not possible that one of the successors of a given MOD M is executed, if the tokens sent by MOD M are not available to the other successors. The consequence of this property in the replication of a module is that all the replicas will receive the same input tokens and the only possibility able to invalidate the sufficient condition expressed in [8] is the different delay associated to the communication of the tokens. To guarantee that messages arrive at the various replicas in the same order it is possible to structure a portion of a BSM program following the strategy of inserting a Sequencer MOD Seq between the predecessors and the replicas of MOD M. MOD Seq is an < ndmod > which receives all the input tokens for the various replicas and forwards these tokens one by one to all the replicas. Since its execution is atomic all the replicas must have received a token before MOD Seq is allowed to forward the next one, thus obtaining the same ordering of the input tokens for all the replicas.

All the replicas of the replication have the following structure:

a) the input channels are defined for the reception of the same tokens that MOD M previously received, with MOD Seq as the sender of these tokens;
b) the output channels allow the forwarding of the same results that MOD M previously forwarded, but the destination of the output tokens is a MOD Adjud whose task is to check these result and to propagate the flow of the computation only if no errors are detected.

As an example let suppose to have a non-deterministic MOD M, as in Fig. 5:

MOD M::= IN {(P_1, a: $type_1$)}
[IN {(P_2, b: $type_2$)} --> $block_1$ OUT{(S_1, d)}
[] IN {(P_3, c: $type_3$)} --> $block_2$ OUT{(S_2, e)}]

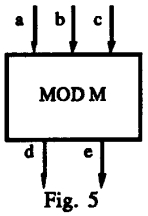

Fig. 5

The structure of the replication is shown in Fig. 6:

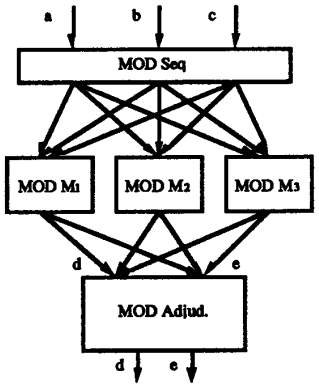

Fig. 6

and the modules are the following:

MOD Seq::=
[IN {(P_1, a: $type_1$)} --> OUT{(M_1, a) (M_2, a) (M_3, a)}
[] IN {(P_2, b: $type_2$)} --> OUT{(M_1, b) (M_2, b) (M_3, b)}
IN {(P_3, c: $type_3$)} --> OUT{(M_1, c) (M_2, c) (M_3, c)}]

MOD M_i::= IN {(Seq, a: $type_1$)}
[IN {(Seq, b: $type_2$)} --> $block_{1i}$ OUT{(Adjud, d_i)}
[] IN {(Seq, c: $type_3$)} --> $block_{2i}$ OUT{(Adjud, e_i)}]

MOD Adjud ::=
[IN {(M_1, d_1) (M_2, d_2) (M_3, d_3)} Adj.Pred. (d_1, d_2, d_3) -->
Adj.Fnct. < choose var-list$_{adj}$ > OUT{(S, var-list$_{adj}$)}
[] IN {(M_1, d_1) (M_2, d_2) (M_3, d_3)}
Not (Adj.Pred. (d_1, d_2, d_3)) -->
OUT{(E.H., d_1) (E.H., d_2) (E.H., d_3)}
IN {(M_1, e_1) (M_2, e_2) (M_3, e_3)} Adj.Pred. (e_1, e_2, e_3) -->
Adj.Fnct. < choose var-list$_{adj}$ > OUT{(S, var-list$_{adj}$)}
IN {(M_1, e_1) (M_2, e_2) (M_3, e_3)}
Not (Adj.Pred. (e_1, e_2, e_3)) -->
OUT{(E.H., e_1) (E.H., e_2) (E.H., e_3)}]

Even if the use of MOD Seq solves the problem of having ordered inputs to all the replicas, another problem exists, related to avoiding divergencies, due to the fact that each replica may start its execution (by verifying the

firing rule) in different times. What is necessary is that all replica start their execution by evaluating the firing rule, with respect to the same set of tokens available in their IN list. Therefore the run time support must evaluate the firing rule for each replica in such a way that between two consecutive evaluation at most one token is made available.

4. Software Fault-Tolerance Techniques

Software fault-tolerance is based on design diversity. Following [2], [6] design diversity can be defined as "the production of two or more systems aimed at delivering the same service through separate designs and realizations". A design based on diversity is composed of a set of variants greater or equal to two and an adjudicator, based on some predefined adjudication strategy, for providing an error-free result from the execution of the variants. The most referred strategies for software fault-tolerance are: 1) recovery blocks [7] , 2) N-version programming [1] and 3) N-self-checking programming [6].

4.1. Recovery blocks

The recovery block approach is based on sequential execution of variants, named alternates, and of an acceptance test on the results provided by each of them. Let us consider the case of a computation composed by one module or an entire subgraph composed of several BSM modules, which we will call MOD M, which we want to structure following the recovery block approach. Of course, the variants must be equi-functional and therefore they must be defined on the same set of input values. This implies that the variants have the same interfaces, that is, the same inputs and the same outputs. In the case MOD M is an entire subgraph, they must be grouped in the first module of MOD M and in the last respectively; the first module therefore will be responsible for the non-deterministic choices of the variant.

An activation of the recovery block requires that the variants are executed in successive times on the same pattern of data; therefore the BSM structuring must guarantee that each variant, in an activation, has available the same data. To this aim an Activator MOD Activ is created an used as interface between the predecessors and the several alternates; MOD Activ provides the data, related to an activation of the recovery block, to the first variant and to a Propagator MOD $Prop_1$ which has to propagate the data to the second variant in case of failure of the first one. The first variant MOD M_1 provides its result to a MOD Chk_1 which implements the acceptance test. MOD Chk_1 in case of successful test sends the results computed by MOD M_1 to a Collector MOD Collect and signals to MOD $Prop_1$ to delete the data MOD $Prop_1$ has previously received; in case of failure of MOD M_1, MOD Chk_1 enables MOD $Prop_1$ to propagate the data to MOD M_2 and to MOD $Prop_2$, and inhibits an input guard in the MOD Collect. MOD Collect must be used for maintaining the correct sequencing of the results of different activations of the recovery block. The need for this functionality of MOD Collect derives from the pipeline which is allowed by our structuring; in fact, it is possible that different variants are active on data related to different activations of the recovery block. The structure of the BSM implementation of the recovery block scheme is in Fig 7.

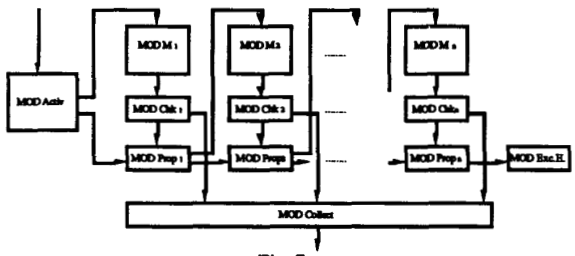

Fig 7

The algorithm for the transformation to be applied is:

1) the variants MOD M_i, i = 1..n, are created; the first module of each of them has in its IN list the necessary input couples; for the first variant MOD M_1 the sender is the interface module MOD Activ, while for MOD M_i the sender is MOD $Prop_{i-1}$ The OUT list of the last module of each variant MOD M_i contains the output couples which name MOD Chk_i as destination;
2) MOD Activ is created; it is of the same type, < simplemod > or < ndmod >, as MOD M and if it is an < ndmod >, it has exactly the same guards of the original MOD M. Its IN list is exactly the same of MOD M and its OUT list list contains for each value received two couples, one with the name of the first variant MOD M_1 and one with the name of MOD $Prop_1$;
3) MOD Chk_i are created, one for each variant MOD M_i; their IN list contain a set of couples from MOD M_i; their OUT list contains a set of couples one naming MOD $Prop_i$ and the others naming the MOD Collect;
4) MOD $Prop_i$ are created, one for each variant MOD M_i; their IN list contain a set of couples from MOD $Prop_{i-1}$ to receive the data of the activation of the recovery block (MOD $Prop_1$ receive this data from MOD Activ) and from MOD Chk_i a Boolean related to the success or failure of the variant MOD M_i. Their OUT list contains a set of couples naming MOD M_{i+1} and MOD $Prop_{i+1}$ (MOD $Prop_n$ will contain couples naming MOD Exc.H., if needed);

5) MOD Collect is created; it is an < ndmod >; its IN lists contain a set of couples from all the MOD Chk_i with the results from the variants and one for each MOD Chk_i with a Boolean used for maintaining the correct sequencing of the results of different activations of the recovery block.

The structure of MOD Activ (assuming MOD M_i as a <simplemod>), MOD Chk_i, MOD $Prop_i$ and of MOD Collect is the following:

```
MOD Activ ::= IN {(P, var-list)}
    OUT{(M1, var-list) (MOD Prop1 , var-list)}

MOD Chki::=
IN {(Mi, var1: type1) ... (Mi, var m: typem)}
[ Pred (var1...varm) --> var bi, go: Boolean;   bi:= true;
  go := false  OUT{(Collect, var1) ... (Collect, varm)
                   (Collect, bi:) (Propi, go)}
[] Not (Pred (var1...varm)) --> var bi, go: Boolean; bi:= false;
  go := true OUT{(Collect, bi) (Propi, go) } ]

MOD Propi::=
IN {(Propi-1, var-list) (Chki, go: Boolean)}
[    go --> OUT{(Mi+1, var-list) (Propi+1, var-list)}
[]   Not (go) --> OUT{ } ]

MOD Collect ::=
[ IN {(Chk1, var1: type1) ... (Chk1, var m: typem)
        (Chk1, b1: Boolean)}    b1 -->
    OUT {(S, var1) ... (S, varm)}
[] IN {(Chk2, var1: type1) ... (Chk2, var m: typem)
        (Chk1, b1: Boolean) (Chk2, b2: Boolean)}
        Not (b1) AND b2 -->
    OUT {(S, var1) ... (S, varm)}
    .....
 IN {(Chkn, var1: type1) ... (Chkn, var m: typem)
        (Chk1, b1: Boolean) (Chk2, b2: Boolean) ...
        (Chkn, bn: Boolean)} Not (b1) AND Not (b2)
        AND...Not (bn-1) AND bn -->
    OUT {(S, var1) ... (S, varm)} ]
```

4.2. N-version programming

The N-versions programming approach is based on a set of variants, named versions, and the adjudicator which performs an adjudication function on all the results provided by the variants. Its implementation is very similar to the replication and in the BSM framework such a technique can be implemented simply by following the algorithms given for the replication. It will not be shown here.

4.3. N-self checking programming

N-self-checking programming is based on the use of a set of N self-checking components. Therefore we will focus our attention on the transformations needed to introduce such components in a computation.

A possible implementation of this technique in BSM consists in substituting the module MOD M to be made self-checking with two replicas, selecting a Distributor MOD Distr or a Sequencer MOD Seq to be used as interface module between the two variants and the predecessors depending on the deterministic or non-deterministic nature of MOD M and by making the results of the two variants compared by a module MOD Compare which compares the results, taking the proper action. The following transformation can be applied to BSM programs if MOD M has to be made self-checking:

1) two replicas MOD M' and MOD M" of MOD M are created; their IN list is contains the same values originally in the IN list of MOD M and the sender module is the interface module MOD Distr or MOD Seq, and their OUT list contains the same variables of the original OUT list of MOD M; these values are sent to MOD Compare;
2) the OUT list of each predecessor module is changed for the interface module MOD Distr or MOD Seq being the receiver of the values previously received by MOD M;
3) the interface module is created, its IN list is the same MOD M had before and its OUT list contains for each value received a couple towards the variants MOD M' and MOD M";
4) MOD Compare is created; its IN list contains all the couples related to the output values from MOD M' and MOD M". The comparison Comp ($var\text{-}list_1$, $var\text{-}list_2$) is performed on the values to be checked and the OUT list of MOD Compare is the original OUT list of MOD M, if a decision is taken on what $var\text{-}list_{adj}$ to propagate;
5) The IN list of the successor modules of MOD M are changed by inserting MOD Compare as sender for the couples in which MOD M was the sender.

The structure of MOD Distr, MOD Seq and MOD Compare is the following:

```
MOD Distr ::= IN {(P, var-list)}
    OUT{(M', var-list1) (M", var-list2)}

MOD Seq::=
[ IN {(P1, vara: type1)}  --> OUT{(M', vara) (M", vara)}
[] IN {(P2, varb: type2)}  --> OUT{(M', varb) (M", varb)}
        .....
 IN {(Pk, vark: typek)}  --> OUT{(M', vark) (M", vark)} ]

MOD Compare ::=  IN {(M', var-list1) (M", var-list2)}
[  Comp (var-list1, var-list2)  --> OUT{(Succ, var-listadj )}
[] Not (Comp (var-list1, var-list2))     -->
    OUT{(Exc.Handler, var-list1) (Exc.Handler, var-list2)} ]
```

Let us consider the example in Fig 8.

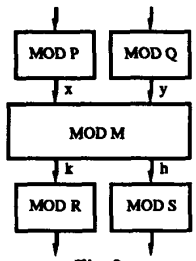

Fig 8

The structure of MOD P, MOD Q, MOD M, MOD R and MOD S is:

MOD P ::= IN{(..,..)} $Block_1$ OUT{(M, x)}
MOD Q ::= IN{(..,..)} $Block_2$ OUT{(M, y)}
MOD M::= IN{(P, x: $type_x$) (Q, y: $type_y$)}
< Compute k and h > OUT{(R, k) (S, h)}
MOD R ::= IN{(M, k: $type_k$)} $Block_3$ OUT{(..,..)}
MOD S ::= IN{(M, h: $type_h$)} $Block_4$ OUT{(..,..)}

Suppose we want to make MOD M self-checking. The structure of the modified module is in Fig.9:

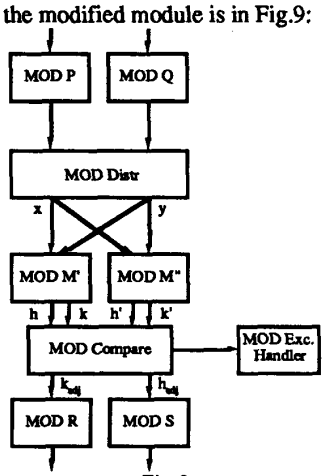

Fig 9

and the modules are the following:

MOD Distr ::= IN {(P, x: $type_x$) (Q, y : $type_y$)}
OUT{(M', x) (M", x) (M', y) (M", y)}

MOD Compare ::= IN {(M', k: $type_k$) (M", k': $type_k$)
(M', h: $type_h$) (M", h': $type_h$)}
[Comp (h, k, h', k') -->
< choose k_{adj}, h_{adj} > OUT{(R, k_{adj}) (S, h_{adj})}
[] Not (Comp (h, k, h', k')) -->
OUT{(E.H., k) (E.H., h) (E.H., k') (E.H., h')}]

MOD P ::= IN{(.., ..)} $Block_1$ OUT{(Distr, x)}
MOD Q ::= IN{(.., ..)} $Block_2$ OUT{(Distr, y)}
MOD M' ::= IN{(Distr, x: $type_x$) (Distr, y: $type_y$)}
< Compute k and h > OUT{(Compare, k) (Compare, h)}
MOD M"::= IN{(Distr, x: $type_x$) (Distr, y: $type_y$)}
< Compute k' and h' > OUT{(Compare, k') (Compare, h')}
MOD R ::= IN{(Compare, k_{adj}: $type_k$)} $Block_3$ OUT{(.., ..)}
MOD S::= IN{(Compare, h_{adj}: $type_h$)} $Block_4$ OUT{(.., ..)}

Other implementation of self-checking components which use a non replicated module with an acceptance test is straightforward.

5. Conclusions

In this paper we have introduced a structured way for inserting software redundancy in BSM programs, and have shown its applicability in dealing with the most referred techniques for software fault-tolerance. It can be understood that the straightforward transformations can be quite easily automatized, by the use of a tool like a precompiler, which receives directives from the programmer on what modules or program sections have to be made redundant and with what type of scheme. This tool can profitably be inserted in a development programming environment, which takes full advantage of the graphical representation of a computation expressed in BSM. Such environment can be based on a graphical interface, similar to the one described in [3].

References

[1] A. Avizienis, "The N-Version Approach to Fault-Tolerant Systems," IEEE TSE, Vol. SE-11, pp. 1491-1501, Dec. 1985.
[2] A. Avizienis and J.C. Laprie, "Dependable Computing: from Concepts to Design Diversity," Proc. of the IEEE, Vol. 74, pp. 629-638, May 1986.
[3] O. Babaoglu, L. Alvisi, A. Amoroso and R. Davoli, "Paralex: an Environment for Reliable Parallel Programming in Distributed Systems", May.
[4] A. Bondavalli and L. Simoncini, "Dataflow-like Model for Robust Computations," Journal of Computer System Science and Engineering, Butterworths, Vol. 4, pp. 176-184, July 1989.
[5] R. Jagannathan and E.A. Ashcroft, "Fault Tolerance in Parallel Implementations of Functional Languages," in Proc. FTCS 21, Montreal, Canada,, 1991, pp. 256-263.
[6] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," IEEE Computer, Vol. 33, pp. 39-51, July 1990.
[7] B. Randell, "Design Fault-Tolerance," in "The Evolution of Fault-Tolerant Computing", A. Avizienis, H. Kopetz and J. C. Laprie Ed., Springer-Verlag, 1987, pp. 251-270.
[8] A. Tully and S. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs," in Proc. SRDS-9, Huntsville, Alabama, USA, 1990, pp. 104-113.