Safety Science, Elsevier. Issue 5, Vol.42, pp.385-403, Juin 2004.

An Investigation on Mutation Strategies for Fault Injection into RDD-100 Models

Mohamed Kaâniche[*], Yannick Le Guédart[*], Jean Arlat[*], Thierry Boyer[**1]

[*]LAAS-CNRS
7 avenue du Colonel Roche, 31077 Toulouse Cedex 4 — France
{Mohamed.Kaaniche, Jean.Arlat}@laas.fr

[**]Technicatome
BP 34000, 13791 Aix-en-Provence Cedex 3 — France

Abstract

This paper focuses on the development of a conceptual framework for integrating fault injection mechanisms into the RDD-100 tool[2] to support the dependability analysis of computer systems early in the design process. The proposed framework combines functional and behavioral modeling, fault injection and simulation. Starting from the RDD-100 model built by the system designers, two techniques are discussed for the mutation of this model to analyze its behavior under faulty conditions: a) insertion of saboteurs into the model, and b) modification of existing component descriptions. Four types of fault models are distinguished and specific mechanisms to simulate the corresponding fault models are proposed for each mutation technique. An approach combining the advantages of both techniques is proposed and a prototype implementing this approach is briefly described.

---

[1] Thierry Boyer is now at Technicatome – Centre d'Études de Saclay, BP 17, 91192 GIF-Sur Yvette Cedex — France

[2] RDD-100 is an industrial tool commercialized by *Holagent Corporation,* USA. RDD stands for "Requirement Driven Development". http://www.holagent.com

1 Introduction

Designing cost-effective fault-tolerant computer architectures is today one of the main concerns for the developers of dependable systems in a large segment of industrial control applications. However, besides evaluations based on probabilistic modeling or FMECA (Failure Modes Effects and Criticality Analysis), the consideration in the early phases of the development process of dependability issues encompassing detailed behavioral analysis is still hardly supported in practice in industry, with few exceptions such as [1, 10]. Accordingly, we have developed a method to assist fault-tolerant systems designers by incorporating the explicit analysis of the faulty behavior of such systems, in the early phases of the development process. The aim is to support designers in making objective choices among different high-level architectural options and associated fault tolerance mechanisms. The proposed approach is based on the development of a functional and behavioral model of the system, and on the behavioral analysis of the model by means of simulation and fault injection.

Several related studies have addressed the issue of supporting the design of dependable systems by means of simulation ([2, 3, 5, 6]). Although these approaches are generally supported by efficient tools, these tools are not designed to be included in the design process. Indeed, for cost and efficiency reasons, system designers would like to use the same formalism and tool to carry out preliminary functional and behavioral analyses, and to assess the effect of faults on the model of the system. This is why we have tried to attack this issue from a different perspective, i.e., to study instead how several existing design tools can be enhanced to support such an early dependability validation analysis. In contrast to the work reported in [1], that dealt with formal techniques (in particular, SDL), we focus here on a more pragmatic approach aimed at elaborating on the modeling and simulation capabilities of system engineering tools used in the industrial world by system designers. Statemate [7] and RDD-100 are among the various commercial tools that are currently used in industry. Mainly based on the preliminary experiments we have been carrying out to assess the suitability of both tools to analyze the dependability of systems in the presence of faults [12], our work is now focused on RDD-100.

This paper builds on our work reported in [9]. It is organized as follows. Section 2 deals with simulation-based fault injection and outlines the main reasons that led us to focus on model mutation. Section 3 summarizes the lessons learnt from the preliminary experiments that we carried out on real-

life case studies. In particular, these experiments highlighted the need to define a generic approach to support model mutation. The feasibility of such an approach based on the RDD-100 tool is discussed in Section 4. Two model mutation techniques are considered: the use of dedicated fault injection components called saboteurs, and code mutation. Specific mutation mechanisms are proposed to implement each technique and are analyzed with respect to the fault models to be simulated. A comparison of both techniques is also provided. Finally, Section 5 provides the conclusion to the paper.

2 Simulation-based fault injection

Starting from the original model (called *nominal model*) built by the designers to carry out preliminary functional and behavioral analysis of the system under nominal conditions, two main approaches can be considered for analyzing fault effects on the system behavior (e.g., see [8]). The first one uses the simulator built-in commands to alter the behavior of the model during the simulation (i.e., without modifying the nominal model). The second one (Figure 1) consists of: a) mutating the nominal model before the simulation by adding dedicated mechanisms for injecting faults and observing their effects, and b) simulating the mutated model to analyze the effects of faults. For both approaches, the analysis of fault effects is carried out by comparing the traces obtained from simulating the system behavior under nominal conditions and faulty conditions, respectively.

The applicability and efficiency of the first approach, with respect to the types of faults that can be injected and their temporal characteristics, strongly depend on the functionalities offered by the command language of the simulator. However, the second approach can take advantage of the full strength of the modeling formalism, i.e., any fault model that can be expressed within the semantics of the modeling language can be implemented. Moreover, this approach is well suited for the definition of generic fault injection mechanisms that can be included automatically in any model and transparently to the user. For these reasons, we have focused our investigations on the second approach.

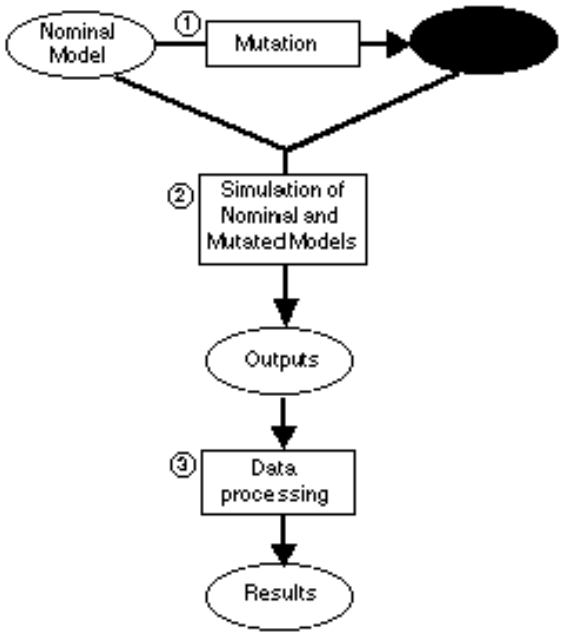

Figure 1 – Fault injection based on model mutation

3 Lessons learnt from preliminary experiments

Several tools are used in industry to support functional and behavioral analysis of computer systems based on modeling and simulation, e.g., Statemate and RDD-100. To study the suitability of these tools for analyzing system behavior in the presence of faults, we carried out several experiments on four real-life case studies [12]. In the sequel, we summarize the characteristics of the four target systems and the main insights gained from this study.

As shown in Table 1, the four target systems were all related to critical applications from distinct fields (nuclear propulsion, ground-based and on-board space systems). The systems architectures were designed to satisfy stringent safety and availability requirements. They include several redundant components using voting and reconfiguration fault tolerance mechanisms.

Table 1: Target systems and experiments

Target system	Dependability requirements	Main architectural features	Aim of the experiment
S1 (submarine nuclear propulsion system control)	*Safety* and *Availability* (a submarine is considered lost should the propulsion system fail)	Recurrent architecture pattern consisting in an active triple modular redundant (TMR) architecture with majority voting (2/3) on the logical outputs.	Study to what extent the simulation approach could help system designers in tuning the functional and structural features of the architecture
S2 (ground-based space system aimed to transmit critical commands to a launcher)	*Safety* (protect populations that could be endangered by the launcher) and *Availability*	A dual redundant architecture with two self controlled channels operating in semi-active mode, and a TMR architecture implementing the configuration module aimed to switch between the two channels.	Assess the ability of the Statemate and RDD-100 tools in handling critical timing issues
S3 (ground-based space system aimed to help navigation among a wide range of users: aircraft, boats, cars,…)	*Availability* (in order to offer a continuous signal to the users)	Five identical processing units (active redundancy). Choice of the operational unit is made by another system using dual redundancy.	Model the algorithm for electing the master processing unit of the system, and observing model behavior upon the occurrence of faults
S4 (on-board space sub-system, devoted to the monitoring and management of reconfigurations for the rest of the system)	*Availability* and *Error confinement* (no propagation of local errors to the rest of the system)	Four identical computing units (hot redundancy) and three identical I/O units (cold redundancy), N-version programming, executable assertion (on-line software error detection)	Analyze potential links, overlaps and complementarities between FMECA and the simulation under faulty behavior method.

In these experiments, we used RDD-100 and Statemate to:

1) model some critical features of each system (e.g., synchronization, reconfiguration management) based on the specification documents available;
2) inject faults into the models (corrupting data, simulating delays and omissions, etc.);
3) analyze the impact of these faults using the simulation engines integrated in the tool.

These experiments confirmed that significant insights can actually be obtained from the analysis of systems behavior in the presence of faults, early in the design process. For example, a model of a real-time ground-based space subsystem (S2 in Table 1) was developed to analyze the impact of temporal faults on the system behavior. This experiment allowed us to reveal an error detection latency problem that was related to the overlapping of two error detection mechanisms and an incorrect sequencing of recovery actions.

From the point of view of fault injection implementation, these experiments highlighted the need to define a set of generic mechanisms allowing the mutation of system models in a systematic way, rather than on an *ad-hoc* basis. Indeed, to be applicable to the analysis of complex real-life systems in an industrial context, the following three requirements should be satisfied by the mechanisms dedicated to injecting faults and observing their effects:

1) The mutated model should preserve the behavior and properties of the nominal model if faults are not injected during the simulation. This will ensure that the mutation mechanisms will not alter the behavior of the nominal model in the absence of faults. Also, the comparison of the simulation traces obtained from the mutated and the nominal models will be made easier, as this comparison can be focused on the instants when faults are injected.
2) Model mutation should be based on the semantics of the modeling formalism, and not on the target system model. The objective is to provide generic mutation mechanisms that are applicable to a large set of systems, rather than to a particular system.
3) Model mutation should be performed automatically and transparently to the user. The latter should be involved only in the specification of the faults to be injected and the system properties to be analyzed.

Finally, it is worth pointing out that the experiments carried out allowed us to identify that in most cases, similar behaviors can be described by the features provided by the respective supporting tools. Still, the use of RDD-100 was found more intuitive than Statemate to describe detailed temporal features. This is mostly due to the fact that the formalism inherits largely from the SDL formalism. Nevertheless, this alone is not enough to draw a clear-cut preference between Statemate and RDD-100.

Thus, although both Statemate and RDD-100 formalims are widely used in industry, and especially by the three industrial partners involved in this preliminary study, the choice of the RDD-100 formalism made by the industrial partner involved in the conduct of further investigations aimed at the definition of an approach targeting the previously identified requirements was simply based on industrial strategic reasons.

Accordingly, our investigations concerned the two following objectives:

1) The definition of model mutation mechanisms based on the RDD-100 formalism.

2) The development of a prototype tool that integrates those mechanisms into RDD-100. The rest of this paper summarizes the main results obtained from our study.

4 Fault injection into RDD-100 models

Two different techniques for model mutation have been investigated. The first one is based on the addition of dedicated fault injection components called "saboteurs" to the RDD-100 nominal model. The second one is based on the mutation of existing component descriptions in the RDD-100 nominal model. Before describing these two techniques, we provide first a summary of the main concepts of RDD-100 formalism.

4.1 RDD-100 main concepts

RDD-100 incorporates a coherent set of formalisms that enable engineers to: 1) define requirements and allocate them to system components according to a hierarchical approach, 2) refine their behavioral description into discrete processes and allocate these to system interfaces, 3) establish system feasibility on the basis of resources and costs, and 4) iterate the engineering design process with increasing levels of detail. Our study focuses on the behavior diagrams formalism (named F-Net) defined within RDD-100 to support the detailed description and simulation of system and components behavior.

A behavior diagram is a graphical representation of the modeled system, composed of a set of concurrent and communicating (*SDL-like*) extended finite-state machines, called *Processes*. The finite-state machine *is extended* in the sense that it has memory and can make decisions about responses to stimuli based upon that memory, and *communicating* in the sense that it can stimulate other processes by sending messages to them. The behavior of each process can be detailed by describing the set of *Functional Modules* executed by the process and their interactions. Functional modules whose behavior can be described hierarchically are called *TimeFunctions* and those that are not further decomposed are called *DiscreteFunctions*. Three types of items can be input to or output from a function: *Global*, *State*, *Message*. Global items are available to all functions within the model. State items are available only to the functions of the same process. Messages can be sent only from

one process to another. Each function can receive at most one message. However, it can send several messages, one per recipient.

Figure 2 presents a simple example of a behavior diagram composed of two concurrent processes: Process 1, composed of module F, and Process 2, composed of module G. Concurrency is represented by the node (&). At the beginning of the model execution, a token is produced by the simulator. The token flow through the model structure describes different execution scenarios. When a token reaches the top node of a parallel branch, the simulator creates a new token for each process. The simulator treats each process concurrently. When the tokens of all processes reach the bottom node (&), the simulator recombines them into a single token and advances the token to the next node on the diagram.

Functional modules execution is initiated when the token and the input messages (if any) reach the module. Then, the data transformations, defined in the functional part of the module (coded in SmallTalk) are processed, i.e., reading and writing data in the state variables and sending messages (if any) to the subsequent output modules. The outputs become available once the module execution time defined in the module code is elapsed.

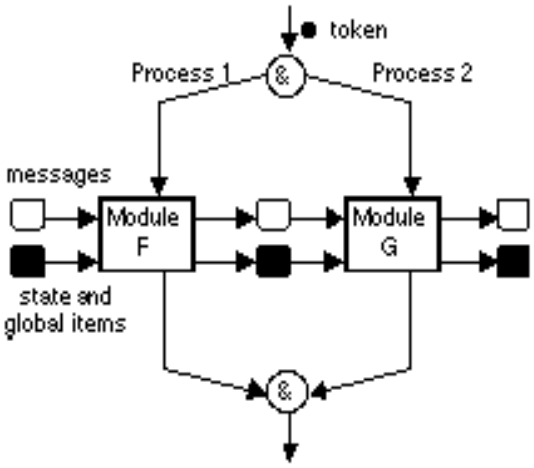

Figure 2 - Example of a behavior diagram

RDD-100 provides three constructs or structures (*Select, Iterate, and Replicate*) allowing for a concise representation of complex behavior diagrams. The Select structure enables to choose among multiple execution paths at the output of a given module depending on conditions specified in the code of that module. The Iterate structure describes the repetitive execution of the sequence of modules that appear between two iteration endpoints. Finally, the Replicate structure allows the representation of equivalent processes by a single abstraction in the model. This structure is particularly useful for the modeling of redundancy in fault tolerant systems.

4.2. Fault models

Fault injection requires the definition of fault types, fault activation time and fault duration. Different types of faults can be injected to alter the value or timing characteristics of behavior diagrams.

In our study, four types of faults are distinguished:

1) data corruption of global items, state items, or messages, that are input to or output from a functional module;
2) delayed execution of a functional module;
3) non activation of a module when triggered;
4) spurious activation of a module; i.e., the module is activated whereas it is not supposed to.

For each fault type, its activation time can be related to the simulation time or to the model state and its duration may be permanent or transient.

4.3. Model mutation based on saboteurs

In this approach, each functional module of the RDD-100 nominal model is considered as a black box. Fault injection is carried out by dedicated components associated to the target module, called saboteurs. The saboteurs intercept the inputs or the outputs of the target module and possibly alter their value or timing characteristics to imitate the behavior of the module in the presence of faults.

In the following, we describe the approach that we defined to mutate RDD-100 models based on the insertion of saboteurs. This approach aims at satisfying the requirements listed in Section 3. We

first illustrate the mutation of a single functional module, then we discuss how this approach can be applied when the nominal model includes special constructs such as Replicate and Select.

4.3.1. Mutation of a single functional module

For each functional module of the nominal model, we associate two saboteurs: S1 intercepts the inputs of the module and possibly alters their value or timing characteristics, and S2 acts in a similar manner on the outputs of the module. The types of faults to be injected in the target module, as well as their activation time and duration are specified in the functional code of the saboteurs. As illustrated in Figure 3, the mutation consists in transforming the target module F, into three parallel processes, corresponding to the execution of S1, F and S2, respectively. The communication between the saboteurs and F is done through message passing. This mechanism enables the synchronization of the functional module with its associated saboteurs. Indeed, module F execution is initiated when S1 terminates, and S2 is activated after the execution of F. Therefore, the input items sent to F (i.e., messages, Global and State items) may be altered by S1 according to the fault model specified in S1, before execution of F. The analysis of the outputs delivered by the module will allow the designers to analyze the impact of the simulated faults on the module behavior as well as on the system. Similarly, faults can be injected on the module outputs to assess to what extent such faults can be tolerated by the system. If the fault activation time is conditioned upon some Global or State items that are not included in the inputs to module F, an additional input to S1 and S2 is added to include these items as illustrated on Figure 3-b. All this process can be performed automatically.

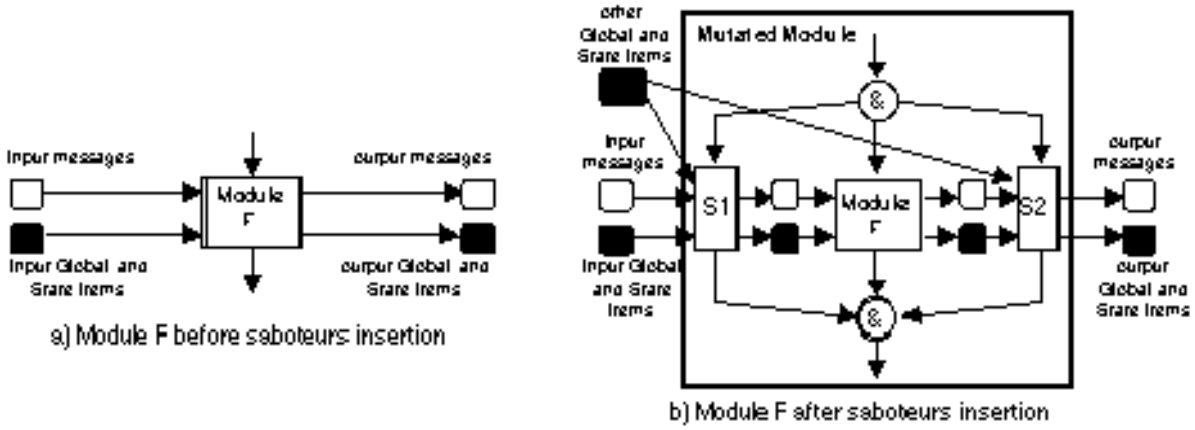

Figure 3- Mutation of a single module

When it is not activated, the behavior of each saboteur is transparent, as if it was not present in the model. The input data are delivered instantaneously to the functional module through S1 without any modification. Similarly, the outputs of the module are made accessible to the model components, through S2. Therefore, the saboteurs remain inactive until a fault is triggered.

4.3.2.Injecting the four types of faults

The injection of the four types of faults defined in Section 4.2 is performed as follows.

- Data corruption can be achieved by substituting new values to the original ones. Any data item or data type defined within a behavior diagram can be modified accordingly.
- A delayed execution of a module is simulated simply by assigning to the saboteur an execution time corresponding to the value of the delay.
- The simulation of the non execution of a module when triggered is done by means of S2. S2 intercepts all the module outputs and ensures that the module remains silent during the duration of the fault.
- The simulation of the spurious activation of a module is more problematic. Indeed, to be activated, a module must receive the activation token as well as the input messages expected by the module (if any). Two solutions can be considered to simulate such a behavior. The first one consists in modifying the behavior diagram to ensure that from each node of the behavior diagram there is a link leading to the target module over which the activation token can flow to activate the target module upon the occurrence of the fault. This solution is not practically feasible, especially when we deal with complex behavior diagrams. Also, it requires a significant modification of the internal code of the functional modules to ensure that the resulting model is consistent. For the second solution, we assume that only the modules of the nominal model that receive input messages may exhibit such a behavior. In this case, the spurious activation of the module can be simulated by means of a specific saboteur, designed to send a message to the target module(s) upon the activation of the fault. This solution is illustrated in Figure 4.

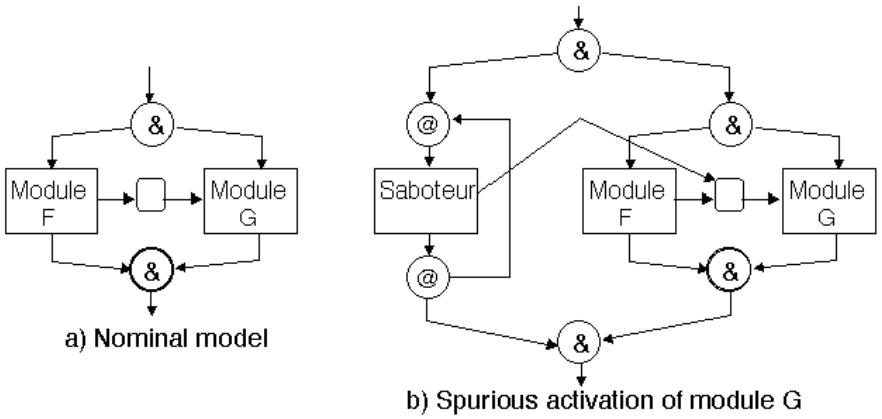

Figure 4- Spurious activation of a module: Example

In this example, the nominal model is composed of two processes corresponding to the activation of modules F and G respectively. To simulate a spurious activation of G at some time *t*, we create a saboteur that is executed concurrently to modules F and G, and remains active during the simulation. This is ensured by using the iterate structure represented by the @ symbol. When the fault activation time is reached, the saboteur sends a message to the target module (G). This message will be taken into account by G when the token is received.

So far, we did not discuss how to mutate the model when some input data are shared by several modules. In this case, various situations can be distinguished; for instance:

1) The fault affects the input interface of one module only.

2) The fault affects the input interfaces of all the modules, and the same error is produced.

3) The fault affects the input interfaces of all the modules, but different error symptoms are produced at each interface; this could correspond for example to the occurrence of Byzantine faults.

The mutation mechanism proposed in Figure 4 reproduces the faulty behavior corresponding to the first situation. Indeed, the saboteurs associated to a given module are designed to alter the execution context of the target module without altering the execution context of the other components of the model. Therefore, if we corrupt an input data that is shared by the target module and other components, the modified input will be propagated to the target module only. The other components

will still perceive the original copy of this input. This choice offers more flexibility to the users with respect to the kind of faulty behavior they would like to simulate. In particular, the faulty behaviors corresponding to situations 2) and 3) described above, can be easily simulated by associating saboteurs to each component, and specifying the attributes of the faults to be injected in the input saboteur according to the faulty behavior to be simulated (i.e., same error patterns for situation 2 and different error patterns for situation 3).

4.3.3. Modules without input or output messages

In Section 4.3.1, we assumed that the functional module to be mutated has input as well as output messages. However, the nominal model might include some modules that do not have messages either in the input or in the output domain. In this case, we just need to create dummy messages, between S1 and the module, or between the module and S2. Once the messages are created, the mutation is performed according to the approach described in Section 4.3.1. It is noteworthy that the content of the dummy messages is irrelevant, as the only role of these messages is to synchronize the execution of the module and its associated saboteurs.

4.3.4. Functional modules with multiple output messages

An RDD-100 module can receive at most one input message. However, it can send multiple output messages, one per recipient. To mutate a module with multiple output messages, we have to adapt the construction proposed in Figure 3 to ensure that no more than one message is sent to the output saboteur, S2. Two techniques can be considered to satisfy this condition. The first one (Figure 5-b) consists in modifying the internal code of the target module to ensure that only one output message is sent to S2. This can be achieved by transforming all the output messages, but one, into Global items. The remaining message will be used to trigger the saboteur S2. When the Global items are accessed by S2, they are transformed into messages before being delivered to their recipients. The second technique consists in associating a saboteur to each output message, and coordinating the execution of these saboteurs as illustrated in Figure 5-c. If the number of output messages to be altered is important, this could lead to very complex mutated models. Clearly, the first solution is more practical, even though its implementation requires the modification of the functional code of the target modules.

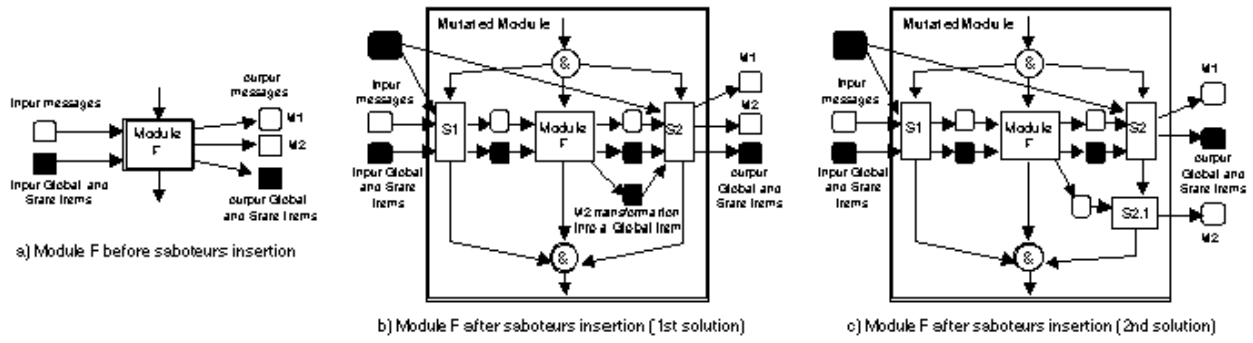

Figure 5- Mutation of a module with multiple output messages

4.3.5 Mutation of a Replicate structure

The Replicate structure is a notation used to specify the simultaneous simulation of multiple copies of the same process within a behavior diagram. It is composed of three parts:

1) A replicate branch defining a sequence of functions that represents a process. This process is treated by the simulator as having multiple copies (replicates) that are simulated concurrently.

2) A domain set defining the number of replicates created when the token enters the replicate structure.

3) A coordination branch controlling the execution of the replicated processes. The coordination branch may send messages to and receive messages from the replicated processes, and may create and delete replicate processes. Any message received by a replicate process must be passed through the coordinate branch.

When the simulator encounters a replicate structure (identified by the node &*), it creates a token for each replicate process, just as when entering a parallel structure. In addition, the token continues on the coordination branch. All processes are simulated concurrently. The replicate structure is exited when all the processes on both branches have terminated. Then, the tokens are rejoined into a single token, which is distributed to the next function or structure on the behavior diagram to be simulated.

As illustrated on Figure 6, two targets can be considered for the mutation of a Replicate structure: the replicated modules and the coordination module. The mutation of each of these targets is

performed according to the process presented in previous Subsections, i.e., two saboteurs S1 and S2 are associated to each mutated module. The mutation of a Replicate structure leads to the replication of the saboteurs associated to each module of the structure. In this context the same faults will be injected in each replica. However, it is also possible to access distinctly each mutated replica, and specify different faults for each mutated module.

The mutation of the coordination module offers to the users several possibilities to alter the global behavior of the replicate structure. Besides altering the content and the timing characteristics of the messages exchanged by the replicas, the mutated coordination module can be used to dynamically remove some replicas from the execution process (e.g., because of failures) or to increase the number of active replicate (e.g., as a result of recovery actions). Thus, this mechanism is particularly useful to analyze the behavior of fault tolerant systems under faulty conditions.

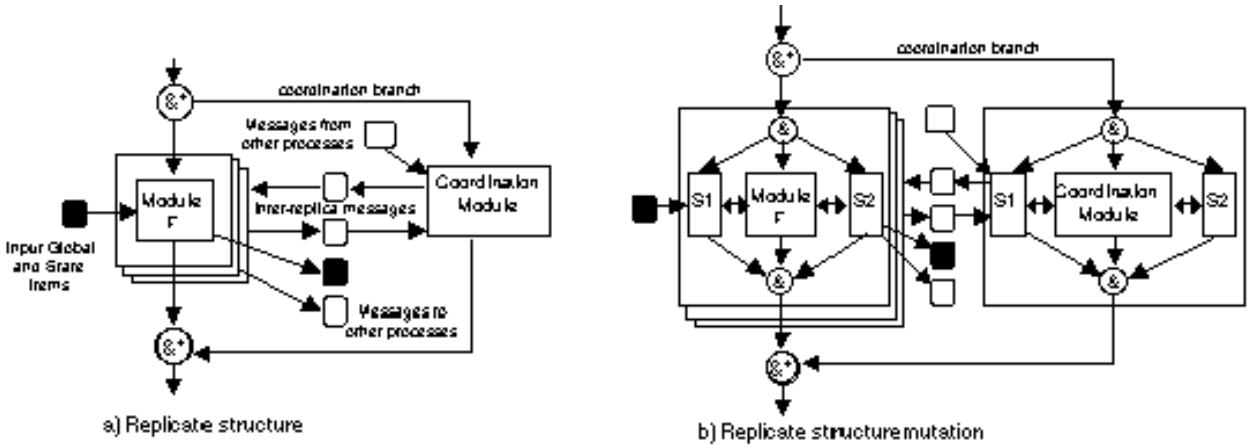

Figure 6- Mutation of a Replicate structure

4.3.6 Mutation of a Select structure

The Select structure is represented by the notation (+) and allows for performing selectively either one process or the other, depending on the arrival sequence of messages from other processes. For example, in Figure 7-a, after the execution of module F, the token may be directed to the branch executing F1 or to the branch executing F2 depending on conditions specified in the F user code.

To support fault injection, each module F, F1 and F2, can be mutated according to the process described in Section 4.3.1. For example, Figure 7-b presents the model obtained when only F is mutated. With this model, any modification of module F inputs remains local to that module and does not affect the inputs of F1 or F2. If one wants to reproduce the same perturbation on the inputs of F1 or F2, two alternatives can be considered:

1) Specify the same kind of fault to saboteurs associated to F1 or F2.

2) Modify the mutated model in Figure 7-b to ensure that any modification of an input item shared by F and the modules involved in the select structure (F1, F2) affects all these modules. This can be done by directing the outputs of saboteur S1 associated with F, to the input interfaces of modules F1 and F2.

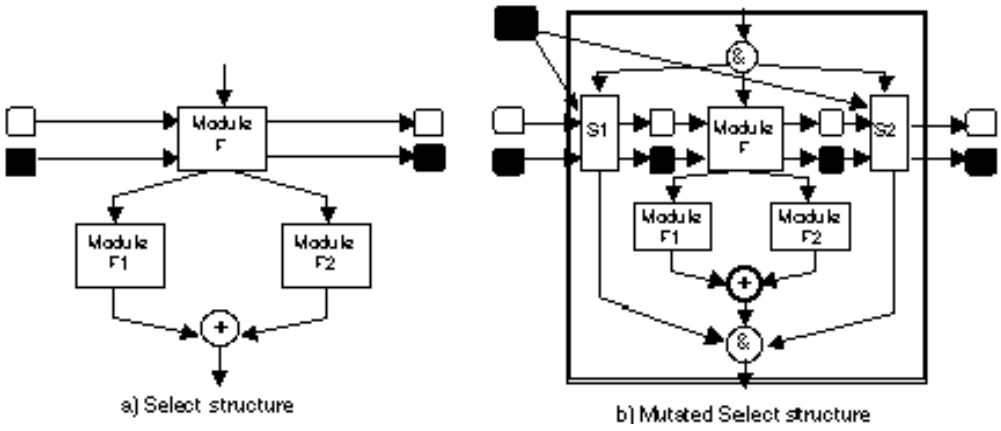

Figure 7 - Mutation of a Select structure

The non activation of a module when triggered can be simulated by ensuring that the outputs of the module are not modified when the fault occurs. If we inject such a fault into module F in Figure 7, its outputs will not be updated when the fault occurs and the token will be directed to either F1 or F2 depending on the system state. Therefore, injecting such a fault does not mean that the whole Select structure is not executed when the fault occurs. If one wants to simulate the latter behavior, it is necessary to modify the internal code of F, i.e., such behavior cannot be simulated simply using saboteurs.

4.4. Code mutation

So far, we assumed that each module of the RDD-100 nominal model was a black box and we analyzed how to mutate the model by inserting saboteurs dedicated to injecting faults at the input or output interfaces of the target modules. In this section we analyze how to mutate the internal code of the nominal model by including the code dedicated to fault injection without inserting additional components into the model.

Code mutation offers a variety of possibilities to alter the behavior of the nominal model. Besides altering the data values and timing characteristics of the target modules, any statement of the original code can be corrupted, e.g., substituting operators or variable identifiers; this is similar to the mutation techniques used by the software testing community [4, 13, 14]. In this paper, we focus only on the fault models defined in Section 4.2.

Data corruption. Any input or output data item (global, state or message) manipulated by a module can be easily accessed. Data corruption consists in substituting to the target items new values corresponding to the faults to be injected. A simple example is presented in Figure 8. The user code of the target module consists in reading two input data items A and B, computing their sum and writing the result C to the output interface (Figure 8a). The mutated code leading to the corruption of A is given in Figure 8b. The output value C can also be altered using the same process. More generally, to reproduce the same faulty behavior implemented with the saboteurs, we have to ensure that when we modify the value of a data item, any occurrence of this data item in the original code is substituted by the new value. This process can be easily automated. Clearly, this technique is easier to implement and is more efficient than the approach based on the insertion of saboteurs discussed in Section 4.3.

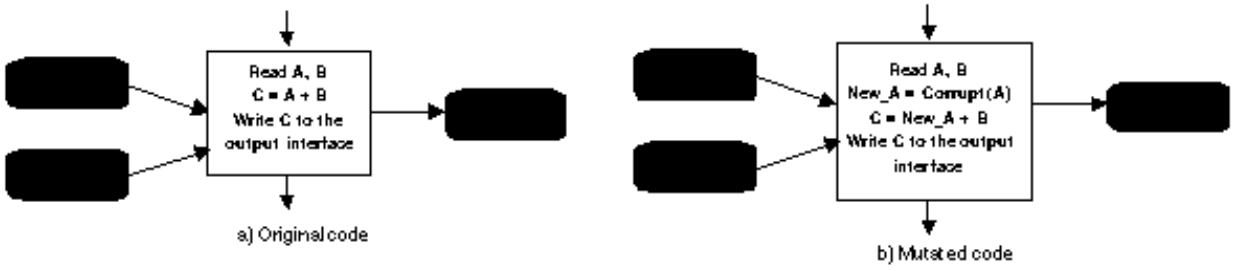

Figure 8 – Data corruption by code mutation

Delays. The simulation of delays can be implemented in two ways: 1) either the execution of the module is delayed by inserting a delay at the beginning of the code, i.e., the input items are read by the target module when the delay specified in the mutated code expires, or 2) the emission of output items to the output interface is delayed. To implement the latter case, we have to identify in the original code each output statement and insert a delay before the corresponding outputs can be accessed.

Nonactivation of a module when triggered. This kind of behavior can be simulated easily by identifying and deactivating all or selected output statements in the target code.

Spurious activation of a module. The simulation of this behavior requires the identification of the modules that send messages to the target module, and then mutate the corresponding code by inserting a statement that forces the emission of a message to the target module when the activation time of the fault is reached. When more than one module is able to activate the target module, an algorithm must be implemented to decide, which module will be selected to send the triggering message when the fault is activated. Clearly, the implementation of this strategy is complex. The solution presented in Section 4.3.1 that is based on the definition of a saboteur dedicated to the activation of the target module appears thus more suitable to simulate this kind of behavior.

Mutation of Replicate and Select structures. The code mutation mechanisms discussed above for a single functional module can also be applied to mutate Replicate and Select structures, with the same advantages and limitations. More details are provided in [11].

4.5. Comparison and discussion

In this section, we analyze to what extent the two mutation strategies presented in Sections 4.3 and 4.4 satisfy the guidelines defined in Section 3.

4.5.1 Preservation of the nominal model properties

The mutation mechanisms based on the saboteurs or on the modification of the RDD-100 components code are designed to be inactive when faults are not injected during the simulation. Considering the first mutation approach, the saboteurs are executed instantaneously and the input data as well as the output data are delivered to their recipients without being altered. If we except the

additional traces resulting from the modification of the behavior diagram structure due to the insertion of the saboteurs, the outputs provided by the simulation of the mutated model and the nominal model will be identical (in the value and timing domain). Therefore, the properties of the nominal model are preserved by the mutated model when it is simulated without activating faults. These properties are also preserved by the mutated model obtained with code components mutation. Note that no additional traces are produced with the latter approach due to the fact that the model structure is preserved.

4.5.2 Independence with respect to the target systems.

All the mutation mechanisms described in the previous sections were defined based on the characteristics of the RDD-100 behavior diagrams formalism and the SmallTalk language used to implement the code of RDD-100 components. They are generic in the sense that they can be applied to any system model built with the RDD-100 tool. Nevertheless, the specification of the types of faults to be injected and of the properties to be analyzed naturally requires a deep knowledge of the target system.

4.5.3 Transparency

Starting from the nominal model, and the specification of the fault injection campaign to be performed, the generation of the mutated model based on the insertion of saboteurs or on the modification of components code can be carried out automatically. A set of mechanisms integrated within the RDD-100 tool can be developed to support the implementation of the mutation strategy. A prototype tool is currently under development to implement this approach [11].

4.5.4 Comparison of the proposed solutions

The two mutation techniques discussed in the previous sections present some advantages and limitations.

Considering saboteurs, the code dedicated to fault injection is separated from the original code. This facilitates the analysis of the mutated model. In particular, the graphical representation of the mutated model clearly identifies the components to be altered as well as their characteristics. Moreover, the saboteurs can be defined as reusable components. Thus, the implementation of model mutation algorithms can be simplified significantly. However, a major problem with this technique is

related to the dramatic increase of the complexity of the mutated model due to the insertion of additional components. As regards the fault modeling capacity of this technique, we have identified some situations that cannot be easily handled simply by using saboteurs and without modifying the original code of the target modules (e.g., functional module with multiple output messages). This is related to the fact that the saboteurs have a restricted view of the target components, i.e., fault injection can be performed only through the input or the output interfaces of these components.

The code mutation technique does not have such limitation. Indeed, any fault model can be simulated, provided it can be expressed within the RDD-100 formalism. Generally, for simple fault models we only need to add a few statements to the target code to support fault injection. Conversely, there are some situations where it is more suitable to use saboteurs to describe the faulty behavior to be simulated. This is the case for example of the simulation of the spurious activation of a target module (See Section 4.4).

Clearly, the above discussion shows that the combined use of saboteurs and code mutation provides a more comprehensive and flexible approach for mutating RDD-100 models. A prototype tool implementing such an approach has been developed. The prototype is implemented in Perl and performs the following tasks (Figure 9):

1) Analysis of the nominal model

2) Set up of fault injection campaign

3) Generation of the mutated model

Tasks 1 and 3 are performed automatically without any interaction with the user.

Figure 9 – Synoptic of the main tasks performed by the prototype tool

The analysis task consists in parsing the textual description of the nominal model and identifying all the model components (processes, TimeFunctions and DiscreteFunctions), the inputs and outputs associated to each component (State items, Global items, Messages) and the interactions between these components (data flow and control flow). The outcome of this analysis is a list of potential targets for fault injection. Based on this list, the user can select the components on which fault injection will be carried out and specify the kind of faults to be injected with their activation time and duration. Based on this specification, the mutated model is automatically generated by the prototype. Code mutation is used to simulate data corruptions, delays, and non activation of modules when triggered. The simulation of a spurious activation of a module is carried out by using a dedicated saboteur. More details about this prototype are given in [11].

5. Conclusion

The main objective of our study consists in integrating fault injection mechanisms into system engineering tools actually used in industry by system designers. By doing so, dependability analyses can be fully integrated early in the development process of dependable systems.

This paper focused on the development of a systematic method for integrating simulation-based fault injection mechanisms into RDD-100 models to support system designers in computer systems dependability analysis. Starting from the functional and behavioral model developed by the designers, the proposed approach consists in mutating the model by including mechanisms aimed at injecting faults, and simulating the mutated model to analyze the impact of injected faults on the system behavior. Two mutation techniques have been studied: the first one consists in adding fault injection components called saboteurs, that are designed to alter the input or output interfaces of the target components, and the second is based on the code mutation of original model components. Four types of fault models have been considered (data corruption, delay, non activation of a module when triggered, and spurious activation of a module) and specific mechanisms have been proposed for each mutation technique to simulate the corresponding fault models. The comparative analysis of techniques showed that a more practical and flexible approach should combine both techniques, instead of using either one or the other. In particular, code mutation can be used to simulate data corruptions, delays, and non activation of components when triggered, while saboteurs are more suitable to simulate a spurious activation of some target components.

Acknowledgments. Yannick Le Guédart was supported by Technicatome. This work was carried out at the Laboratory for Dependability Engineering (*LIS*). Located at LAAS, *LIS* was a cooperative laboratory gathering five industrial companies (Astrium France, Airbus France, Électricité de France, Technicatome, and THALES) and LAAS-CNRS.

References

[1] Ayache S., Humbert P., Conquet E., Rodriguez C., Sifakis J. and Gerlich R., "Formal Methods for the Validation of Fault Tolerance in Autonomous Spacecraft", in Proc. 26th IEEE International Symposium on Fault-Tolerant Computing (FTCS-26), pp.353-357, IEEE Computer Society Press, Sendai, Japan, 1996.

[2] Boué J., Pétillon P. and Crouzet Y., "MEFISTO-L: A VHDL-based Fault Injection Tool for the Experimental Assessment of Fault Tolerance", in Proc. 28th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-28), pp.168-173, IEEE Computer Society Press, Munich, Germany, 1998.

[3] Clark J. A. and Pradhan D. K., "REACT: A Synthesis and Evaluation Tool for Fault-Tolerant Microprocessor Architectures", in Proc. Annual Reliability & Maintainability Symp., pp.428-435, IEEE Computer Society, 1993.

[4] DeMillo R. A., Lipton R. J. and Sayward F. G., "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer Magazine, 11 (4), pp.34-41, 1978.

[5] Ghosh A., Johnson B. W. and Profeta III J. A., "System-Level Modeling in the ADEPT Environment of a Distributed Computer System for Real-Time Applications", in Proc. 1st IEEE International Computer Performance and Dependability Symposium (IPDS'95), pp.194-203, IEEE Computer Society, Erlangen, Germany, 1995.

[6] Goswami K. K., Iyer R. K. and Young L., "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", IEEE Transactions on Computers, 46 (1), pp.60-74, 1997.

[7] Harel D., Lachover H., Naamad A., Pnueli A., Politi M., Sherman R., Shtull-Trauring A. and Trakhtenbrot M., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", IEEE Transactions on Software Engineering, 16 (4), pp.403-414, 1990.

[8] Jenn E., Arlat J., Rimén M., Ohlsson J. and Karlsson J., "Fault Injection into VHDL Models: The MEFISTO Tool", in Proc. 24th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-24), pp.66-75, IEEE Computer Society Press, Austin, TX, USA, June 1994.

[9] Kaâniche M., Le Guédart Y., Arlat J. and Boyer T., "An Investigation on Mutation Strategies for Fault Injection into RDD-100 Models", in Proc. 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2001), (U. Voges, Ed.), Lecture Notes on Computer Science LNCS 2187, pp.130-144, Springer-Verlag, Budapest, Hungary, 2001.

[10] Kaâniche M., Romano L., Kalbarczyk Z., Iyer R. and Karcich R., "A Hierarchical Approach for Dependability Analysis of a Commercial Cache-based RAID Storage Architecture", in Proc. 28th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-28), pp.6-15, IEEE Computer Society Press, Munich, Germany, 1998.

[11] Le Guédart Y., Functional and Faulty Behavior Analysis with RDD-100 — Method and Prototype, LAAS Report, N°00561, 2000 *(in French).*

[12] Le Guédart Y., Marneffe L., Scheerens F., Blanquart J. P. and Boyer T., "Functional and Faulty Behavior Analysis: Some Experiments and Lessons Learnt", in Proc. 29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29), pp.348-351, IEEE Computer Society, Madison, USA, 1999.

[13] Thévenod-Fosse P., Waeselynck H. and Crouzet Y., "Statistical Software Testing", in Predictably Dependable Computing Systems, (J.-C. L. B. Randell, H. Kopetz, B. Littlewood, Ed.), pp.253-272, Springer-Verlag, Berlin, 1995.

[14] Voas J. M. and McGraw G., Software Fault Injection — Inoculating Programs Against Errors, John Wiley & Sons, Inc., 1998.