Fault Injection for Formal Testing of Fault Tolerance

Dimiter Avresky, Member IEEE
Texas A&M University, College Station
Jean Arlat, Member IEEE
LAAS-CNRS, Toulouse
Jean-Claude Laprie, Member IEEE
LAAS-CNRS, Toulouse
Yves Crouzet
LAAS-CNRS, Toulouse

Key Words — **Fault-tolerance testing, Fault/error injection, Design/implementation fault removal, Execution tree, Well-defined formula**

Summary & Conclusions — **This study addresses the use of fault injection for explicitly removing design/implementation faults in complex fault-tolerance algorithms & mechanisms (FTAM), *viz*, fault-tolerance deficiency faults. A formalism is introduced to represent the FTAM by a set of assertions. This formalism enables an execution tree to be generated, where each path from the root to a leaf of the tree is a well-defined formula. The set of well-defined formulas constitutes a useful framework that fully characterizes the test sequence. The input patterns of the test sequence (fault & activation domains) then are determined to cover specific structural criteria over the execution tree (activation of proper sets of paths). This provides a framework for generating a functional deterministic test for programs that implement complex FTAM.**

This methodology has been used to extend a debugging tool aimed at testing fault tolerance protocols developed by BULL France. It has been applied successfully to the injection of faults in the inter-replica protocol that supports the application-level fault-tolerance features of the architecture of the ESPRIT-funded Delta-4 project. The results of these experiments are analyzed in detail. In particular, even though the target protocol had been independently verified formally, the application of the proposed testing strategy revealed two fault-tolerance deficiency faults.

1. INTRODUCTION

Acronyms[1]

FT	fault-tolerance
ftd	FT deficiency
FTAM	FT algorithms & mechanisms
FT/TS	the FT test strategy proposed in this paper.

According to the now widely recognized dependability terminology [1, 2] and as illustrated in [3], fault injection encompasses both objectives of the dependability validation process: fault removal and fault forecasting.

Concerning the *fault removal* objective, fault injection is explicitly aimed at reducing, by *verification*, the presence of faults in the design/implementation of FTAM whose consequences would be *deficiencies* in their anticipated behavior when faced with faults that are explicitly intended to handle. That is, faults are injected to uncover such potential ftd-faults and thus, to determine the most appropriate actions for improving the FTAM.

For *fault forecasting*, the main issue is to rate, by *evaluation*, the *efficiency* of the *operational behavior* of the FTAM. This type of test is aimed mainly at estimating the parameters that usually characterize the operational behavior of the error processing and fault treatment: *coverage* and *latency*.

Much work has been devoted to fault injection [4-10]. Until recently, most experiments on actual fault-tolerant systems have dealt mainly with the fault-forecasting objective. We have shown [11] that the data gathered during experiments aimed at fault forecasting can be used in a feedback loop, to impact the design/implementation of the FTAM. However, this extrinsic fault-removal capability corresponds only to a by-product and is in no way a systematic approach for testing complex FTAM.

Usually, fault-injection based attempts to validate (verify & evaluate) FTAM that consist of test sequences where the input patterns (the injected faults) are selected according to fault/error models intending to simulate (as much as possible) the consequences of activating real faults. Heavy ion radiation [12], pin-level fault injection [7], software-implemented fault injection [13-15], failure acceleration [6], or as well, fault injection in simulation models [10, 16] are typical techniques to reach this objective in the context of physical- or software-fault-injection experiments. See also [17, 18] for surveys on these fault-injection techniques.

As expressed in [3], more efficient & direct approaches to this problem need to be investigated. Indeed, real fault-tolerant system correspond to distributed computing architectures that incorporate a variety of (hardware & software) error detection and error-processing techniques. Although in several cases, elementary error detection mechanisms are simple, their interconnection can pose serious testing problems. Furthermore, the error processing and application-level FT features usually rely on complex software-implemented FT protocols that need to be tested thoroughly. This need has also been recognized independently in [19] which describes the FT testing techniques used for the FAA Advanced Automation System, and in [20] and, subsequently, in [21] which proposes heuristics for deterministically testing FT protocols.

A detailed representation of such complex FTAM and, in particular, of their structure is needed to improve the efficiency of a fault-injection test sequence. Indeed, this would enable the test criteria to be extended to include structural criteria establishing explicit conditions for determining the *input* patterns (from both *fault* & *activation* attributes) that activate the FTAM.

This knowledge then can provide a useful basis for guiding the fault-injection process, in particular for:

[1]The singular & plural of an acronym are always spelled the same.

- monitoring the coverage of the test sequence with respect to the extended test criteria and, thus,
- favoring the detection of potential *ftd-faults* (*design/implementation faults* in the FTAM).

Such guidelines are of prime interest for controlling a fault-injection test sequence, because the number of faults that can be injected in practice is limited. As such, the test should restrict the number of *redundant* experiments (injected faults that lead to an equivalent sensitizing of the FTAM).

Here, we address the problem of efficiently testing FTAM that feature the kind of functions implemented by FT *protocols* (diagnosis, decision-making, and reconfiguration) in a distributed computing environment. We elaborate on work previously reported in [22]. In particular, we report on an enhanced set of experiments performed when applying the FT/TS on a real FT protocol; the presentation & analysis of the results obtained have been extended accordingly.

Section 2 presents the motivation for using a formal testing approach. Sections 3 & 4 present the formalism and its application, respectively, to the testing of FTAM by way of fault injection. Section 5 applies FT/TS to an actual FT protocol, *viz*, the Inter Replica protocol (IRp) of the Delta-4 fault-tolerant distributed architecture [23]. Section 6 describes the FT/TS aimed at providing a general framework suitable to define, construct, and conduct fault injection-based test sequences specifically aimed at removing ftd-faults. Section 7 discusses the results and their application.

2. MOTIVATION FOR USING A FORMAL TESTING APPROACH

The choice of:

- a description level,
- a formalism,
- the derivation of the model,
- the model's processing,

for determining the input patterns of the test are important issues.

The description level depends on the phase, during the development process of the FTAM, when the fault-injection test sequence is to be performed. In early design phases, only high-level functional information is available; however, the expression of the requirements constitutes a valuable framework for specifying the predicates that characterize the service of the FTAM to be used for testing in subsequent phases. The more implementation details that are available, the more structural features that can be included in the model, and the richer are the conditions characterizing the control patterns. However, in practice, the complexity of the FTAM considered requires — even when the implementation is available — that the description:

- be an abstraction of the implementation that can be formally expressed,
- is compatible with the predicates characterizing the service of the FTAM.

The benefits anticipated from this approach come from two distinct issues:

- The specific simplicity of the service provided by most FTAM (tree-like structure, strong dependence on fault set, and limited dependence on activation set [20]).
- The ultimate goal is to characterize as precisely as possible a fault injection test sequence to be performed on the complete FTS. ◄

Potential candidate models for conducting such a test include graphical or procedural expressions of the structure and/or behavior of the FTAM. Typical examples are trees, Petri nets, flow charts, state machine languages, and Calculus of Communicating Systems (CCS-based languages: Estelle & LOTOS languages [24]). The formalism to be used for describing the FTAM must:

- provide facilities to access, either directly or implicitly, the *path* features needed to develop structural testing strategies.
- must permit expression of both the specification of the service and the details abstracted from the implementation, in order to conduct the test. ◄

A model that satisfies many of the requirements for developing structural testing strategies is a *program path tree*. In [25], the tree nodes represented the structures or instruction blocks, and tree edges denoted the temporal relationship existing between the nodes. We have shown that this type of path tree offers excellent opportunities for accessing either directly or implicitly the path features needed to develop structural testing.

However, to overcome the problem of the complexity of the program implementing the FTAM and to provide the designer with possibilities for dealing with the design faults by means of modifying the model of the FTAM (the set of assertions), a formalism has been chosen for obtaining the model of the FTAM and the tree of all possible executions: the *execution tree*.

This formalism is equivalent to state transition models or Petri nets; however, it more easily supports design changes and also features consistency & completion checks of the model. The process of constructing an execution tree can be related to the construction of a *proof tree* in symbolic execution approaches [26]. However, FT/TS is different, as it encompasses both symbolic execution of the model and testing of the actual implementation of the protocol software. Section 3 defines this formalism in detail.

3. DEFINITION OF THE FORMALISM

Acronyms & Abbreviations

WDF	well-defined formula
A-WDF	activated WDF
ET	execution tree
EV	event variable

A-EV	activated EV
AA	atomic action
SAA	sequence of AA (involves more than 1 AA)
SAA*	an AA or SAA (this is used for simplicity of notation)
IEV	initial EV
I-IEV	installed initial event variable: an edge
TERM	TERMinating event variable.

Notation

e	set of basic elements (section 3.1)
r	set of rules forming WDF (section 3.2)
a	set of assertions (section 3.3)
d	set of derivation rules (section 3.4)
ψ	output predicate associated with the event variable labeled TERM
f_j	an injected fault.
$\Rightarrow$	implies: logically follows in time
$\Leftrightarrow$	implies: as a result may come one of the following
&	AND
$\vee$	OR
$\neg$	NOT
IP($\cdot$)	Input_Predicate(Pre-Condition)
PEAT($\cdot$)	Possible Events on Action Termination = Output_Predicate(Post-Condition).

Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

The quadruple $\langle e, r, a, d \rangle$ is defined as a formalism.

3.1 Set of Basic Elements

The set e consists of the subsets:

- EV;
- SAA*;
- logical connectives: &, $\vee$ (OR), $\neg$ (NOT);
- left & right parentheses: (,);
- $\Rightarrow$, $\Leftrightarrow$;
- dedicated EV: IEV and TERM.

An EV is a variable that modifies its value after occurrence of an event. An event can be internal or external to the program. An example of an internal event is the completion of an SAA*, which corresponds to the execution of some procedure(s) in the program. Examples of external events are received messages and injected faults. An EV can be an observable parameter of the system that can have two states, FALSE and TRUE.

IP($\cdot$) establishes the conditions for activating the SAA*.

PEAT($\cdot$) defines a set of EV, among which one might become TRUE after the completion of the SAA*.

An AA is indivisible in the sense that, once initiated, it is carried out to completion. SAA is a more complex action. An SAA* is performed by some procedure(s) that can change the status of *only one* EV $\in$ PEAT from FALSE to TRUE, or can leave all EV unchanged. The EV whose status is to be changed depends on the procedure(s).

3.2 Set of Rules

The set r of rules (R1 - R4) forms a WDF.

R1. Each WDF starts with IEV.

R2. A WDF is a sequence of alternating EV and SAA*, connected by "$\Rightarrow$".

R3. Each WDF terminates with TERM.

R4. The semantics of the WDF describes the behavior of the system under design, according to its specification:

$$\text{IEV} \Rightarrow \text{SAA}_i \Rightarrow \text{EV}_j \Rightarrow \text{SAA}_{i+1} \Rightarrow \text{EV}_{j+1} \Rightarrow \ldots \Rightarrow \text{TERM}. \quad (1)$$

Eq (2) is a *section*:

$$\text{EV}_j \Rightarrow \text{SAA}_i^* \Rightarrow \text{EV}_{j+1}. \quad (2)$$

An example of a section is:

$$\text{C4} \Rightarrow \text{COMPLEX ACTION-1(vote)} \Rightarrow \neg\text{C1},$$

$\neg$C1, C4 are branching variables

COMPLEX ACTION-1(vote) is a complex action.

Each WDF can be considered as a *chain* of sections.

3.3 Set of Assertions

The formal specification of the system is a set of assertions a of the form:

$$\text{IP}(\text{EV}_1, \text{EV}_2, \ldots, \text{EV}_m) \Rightarrow \text{SAA}_i^* \Leftrightarrow \text{PEAT}(\text{EV}_{i,1}, \text{EV}_{i,2}, \ldots, \text{EV}_{i,n}). \quad (3)$$

An assertion is *activated* when its IP($\cdot$) is TRUE. Several criteria for the set of assertions a are postulated as:

A1. the set a is a formal description of the specification;

A2. it must be *complete*, in the sense that a WDF could be derived in the formalism $<e, r, a, d>$, *ie*, there are no missing-assertions (a successor exists for every EV);

A3. it must be *consistent*: only one assertion can be activated at a given time.

Example 3-1

The #a & #b are 2 assertions:

a. IP(ACK & WRRAVL1) $\Rightarrow$ STOP RR $\Leftrightarrow$ C2 $\vee$ $\neg$C2,
b. IP(ACK & WRRAVL1 & C2) $\Rightarrow$ cnf + $\Leftrightarrow$ VALIDATED -TERM;

- C2 is a branching variable,
- ACK is an external EV.
- WRRVAL1 and VALIDATED are internal EV that correspond to certain states of the model.

For the state transition model, IP(·) of the assertion contains a state variable. The following chain of sections is activated by executing sequentially assertions #a & #b:

WRRAVL1 ⇒ CHECK EV.DEV ⇒ ACK ⇒ STOP RR ⇒ C2 ⇒ cnf + ⇔ VALIDATED-TERM. ◄

Example 3-2

The #c is an example of an assertion (generated) by the interpreter where the IP(·) gathers all edges leading to the same action cnf-(MI) abort and provides the same output post-condition KILLED (TERM).

c. IP(C1 & ¬C2 & TURN & (WRRVAL1 ∨ WTOKEN) ∨ C1 & ¬C2 & CLAIM & (WRRVAL1 ∨ WTOKEN) ∨ ACK & ¬C2 & (WTOKEN ∨ WRRVAL1 ∨ WCPVAL1) ∨ LOCAL REQUEST & ¬C2 & (WSIGN0 ∨ WACK0 ∨ WREQ) ∨ CLAIM & C1 & ¬C2 & WCPVAL1) ⇒ cnf-(MI), abort ⇔ KILLED-(TERM);

- C1, C2 are branching variables,
- TURN, CLAIM, ACK, LOCAL REQUEST are external EV,
- WRRVAL1, WTOKEN, WCPVAL1, WSIGN0, WACK0, WREQ are internal EV that correspond to certain states of the model. ◄

3.4 Set of Derivation Rules

Consider a WDF that is valid with regard to the anticipated behavior of the FTAM. The set *d* gives the rules (D1 - D6) for this WDF to be derived from the set of assertions *a*.

D1. The WDF is processed step-by-step from the beginning (IEV) to the end (TERM). At every step, one section is executed as depicted by (2).

D2. At each step of formal derivation, *a* is searched to find an assertion for which:

- the IP(·) of the assertion is true (the assertion is activated);
- the SAA* in the activated assertion corresponds to the SAA* in the current section of the WDF;
- the PEAT(·) of the activated assertion includes the post-conditions of the current section of the WDF.

D3. If such an assertion (in D2) does not exist, then *a* is *incomplete*.

D4. If there exists more than one assertion that is activated, then *a* is *inconsistent.*

D5. If only one assertion is activated, proceed to D6.

D6. The formal derivation continues if the post-condition of the current section is not TERM. The derivation is successful when TERM is reached. ◄

Such a WDF thus can be considered as a *test case* for *a*.

3.5 Execution Tree

Definition: An ET is a tree graph with EV as edges and SAA* as nodes. Only one edge (IEV) extends from the root; edges, labeled TERM, lead to the leaves of the tree.

Table 1 shows the algorithm for building the ET. During the automatic generation of the ET, if no assertion is activated, it means that *a* is not *complete* (there are missing assertions) or if more than one assertion is activated, then *a* is not *consistent*. The missing assertions and the contradiction between assertions require *a* to be checked. In such cases, some revisions should be made by the system designer to complete the generation of the ET.

4. FORMAL TESTING OF FTAM BY FAULT INJECTION

The ET obtained by the algorithm of table 1 can be used as a general structure to represent the behavior of the FTAM.

Table 1. Algorithm for Generating an Execution Tree

```
begin
  IEV := "TRUE"
  while (there exist non-terminated edges in the tree)
    EDGE:=one of the nonterminated edges;
    if (there does not exist an activated assertion)
    /* The set of assertions is "incomplete" */
      STOP
      endif
    if (only one assertion is activated)
      Append the SAA* of the assertion as node to the EDGE;
      Append to the new node all EV, listed in PEAT of the assertion, as non-
terminated edges;
      endif
    if (more than one assertion is activated)
    /* The set of assertions is "inconsistent" */
      STOP
      endif
    endwhile
  end
```

The set of all paths from the root to all leaves is the complete set of WDF, derived from *a* and of the algorithm in table 1. This represents the global knowledge of the system designer. The ET is a *language-independent* notation of the program implementation of the FTAM. It supplies, in a concise and easily processable manner, the characteristics of the program that are useful for developing a suitable efficient testing strategy.

The *testing strategy* for FTAM is based on the set of WDF (the set of all paths from the root to each leaf of the ET). This set represents the *structural deterministic* test for the model of the FTAM and provides a framework for generating a *functional deterministic* test for the program implementing the FTAM.

FT/TS, based on the ET, presents a framework for building an exhaustive deterministic test. The set of WDF allows one to determine the *probe insertion* points [20] on the paths for *monitoring* their activation during dynamic testing [27] and, consequently, for checking the *coverage* of a selected path with the test cases.

Apart from the ET, a WDF can be written to describe the *service* of the FTAM. The derivation of such a WDF on the

set of assertions could be checked by using the set of rules in section 3.4. A similar approach was used [28] for verifying protocol specification, where formulas defining the service were directly evaluated on a state graph. However, FT/TS extends such an approach because:

- it provides stronger conditions for testing the model of FTAM
- it is aimed at deriving cases for testing the program implementation of the FTAM.

4.1 Formal Characterization

The information from the ET can be used for testing the *program implementation* of the FTAM. The synchronization of an external event with the instant when the program is in a specific state can be achieved by applying the appropriate sequence of input data (in the activation set A for the FTAM), which can be obtained from the ET. The considered external event can be a message that can contain an error. An erroneous message is simulated by injection of a fault from the set $\boldsymbol{F}$ of the FTAM, along with the provision of proper activation domain. A non-erroneous message is simply simulated by sensitizing the proper variables. As a result (and for clarity) erroneous external events are called *faults*. In what follows, we focus on the case: the considered event corresponds to the injection of a f_j from $\boldsymbol{F}$. An analogous behavior can be described, and distinct paths can be activated, when no fault is considered.

Definitions (for activation of a specific path in the ET).

- A-Chain. The chain that provides the activation domain to reach a specific edge in the ET; an A-Chain begins from the root of the ET and ends with edge I-IEV.
- A-WDF. It begins with A-EV and ends with TERM, where A-EV is initiated by an external event that triggers from FALSE to TRUE (*eg*, f_j in the case considered here).

The remainder of this section illustrates the relationship between I-IEV and A-EV corresponding to f_j:

$$\text{I-IEV} \Rightarrow \text{AA}_k \text{ (resp. } \text{SAA}_k^*) \Rightarrow f_j. \tag{4}$$

Consider a valid (terminating) path in the ET expressed by the WDF:

$$\text{I-EV} \Rightarrow \text{SAA}_i^* \Rightarrow \ldots \Rightarrow \text{I-IEV} \Rightarrow \text{SAA}_k^* \Rightarrow f_j \Rightarrow \text{SAA}_m^* \Rightarrow \ldots \Rightarrow \psi\text{-TERM}. \tag{5}$$

Lemma 1. There is no *implementation fault* in the path p of FTAM iff it terminates and provides the same ψ as the WDF from the corresponding path of the ET — as depicted by (5):

$$\text{if } (\phi_P \wedge f_j) \text{ then there exists } p \text{ such that } (H_P \wedge \psi). \tag{6}$$

Notation

ϕ_P	program activation domain that corresponds to the A-Chain (I-EV, I-IEV) plus the subsequent SAA_k^* in (5)
f_j	fault that is injected in the program at the instant, when the program state corresponding to the I-IEV of the ET, is reached
p	path that starts from the injected fault f_j and corresponds to the A-WDF of (5)
H_P	proposition that the program terminates
ψ	output predicate. ◄

The proof of lemma 1 relies on the fact that the program implementation of the FTAM is obtained *s*-independently from the ET. As a result, it must have the same behavior (it executes the same actions by means of different procedures, setting up the branching variables, *etc*). It must terminate and produce the same ψ, as predetermined by the ET. The input data enabling proper activation of the program are obtained from a static analysis of the A-Chain and from the A-WDF, *viz*, the sequence of sections starting from f_j and ending with ψ-TERM in (5).

This strategy also applies for testing parts of the FTAM that do not deal with fault-tolerance issues as shown in lemma 2.

Lemma 2. If no fault exists (*ie*, no fault has to be injected), then f_j is simply replaced with A-EV in the WDF of (5); and (6) of lemma 1 is modified as:

$$\text{if } (\phi_P \wedge \text{A-EV}) \text{ then there exists } p \text{ such that } (H_P \wedge \psi). \tag{7}$$

4.2 Practical Outcome

The assessment of (5) & (6) forms a *syndrome* that fully determines the outcome of the test. Table 2 depicts the possible combinations for this syndrome and characterizes the conclusions that can be drawn. Column #1 simply indicates whether the ET may or may not terminate. A non-terminating ET means that the set of assertions is not complete and consistent, as can be revealed by the algorithm in table 1. Nevertheless, considering such a case in the analysis of the syndrome is of practical interest when only a part of the ET is available to conduct the test sequence, as explained in the remainder of this section. Column #2 indicates whether the executed program terminates properly and provides the same output predicate predetermined by the ET.

Table 2. Syndrome analysis

Activated Paths in Execution Trees terminate	Program terminates correctly	Conclusion
yes	yes	Correct activated paths
yes	no	Implementation fault
no	no	Design fault
no	yes	Eliminated design fault

- Row #1 is obtained when (5) & (6) are satisfied.
- Row #2 identifies an *implementation fault* that corresponds to "(6) is not valid (*ie*, the program does not terminate or, if it terminates, it does not supply the same output predicate)".
- Row #3 corresponds to "(5) is not valid"; in most cases, this implies "(6) is not valid either". As a result, a *design fault*

has been revealed. The designer should revise the set of assertions to terminate the path. In some cases, even when (5) is not valid, (6) could be valid; this corresponds to "the design fault might have been eliminated during the implementation of the FTAM". This is identified in row #4 by the term *eliminated design fault*. Such a singular case was observed in the preliminary experiments based on a not-entire execution tree. Otherwise, and according to the general framework, it should be classified as a design fault.

Lemma 1 provides a formal framework for determining $\boldsymbol{F}$ for testing the program that implements the FTAM and for solving the main problem of the synchronization of the injected fault with a specific state of the program during its execution.

The ET provides a template to investigate several structural test criteria (*eg*, coverage of leaves, paths), enabling one to estimate the *structural coverage* of the ET. The selected criteria guide the selection of $\boldsymbol{A}$ & $\boldsymbol{F}$ in the ET. Once $\boldsymbol{A}$ & $\boldsymbol{F}$ have been obtained, they can be applied to test the program. It is not necessary to trace the activated path in the program for testing it. Only ψ has to be observed & checked. Thus, a *deterministic functional test* for the program can be generated be lemmas 1 & 2.

Sections 4.2.1 & 4.2.2 concern the coverage of the program provided by the test.

4.2.1 Coverage Assessment

The coverage can be assessed only from the structural coverage of the ET (generally, the coverage of the ET is providing an *upper bound* for the coverage of the program).

4.2.2 Functional Test

Notation

$\text{PredP}_T(f_j)$ *set of predicted paths* in the ET departing from f_j and terminating with the same output predicate

$\text{PredP}_P(f_j)$ *set of predicted paths* in the program departing from f_j and terminating with the same output predicate.

A deterministic functional test provides only limited power for covering the structure of the program, but its combination with *statistical structural testing* might improve the coverage. Indeed, the functional test of the program does not enable one to identify the path actually tested within the corresponding $\text{PredP}_P(f_j)$. Nevertheless, because:

- $[\text{PredP}_T(f_j), \text{PredP}_P(f_j)]$ is generally appreciably reduced compared to $[P_T(f_j), P_P(f_j)]$ of *all paths* in the [ET, program] departing from f_j; and
- the oracle-problem (how to determine the correct response of the program for each input) is basically solved by knowledge of the ET;

it follows that the structural statistical testing methodology [29] might be used to improve the coverage of the paths within $\text{PredP}_P(f_j)$.

5. APPLICATION

5.1 Target System

FT/TS was applied to the Inter-Replica protocol (IRp) of the Delta-4 distributed fault-tolerant architecture [30], whose specification had been *s*-independently verified formally. The IRp implements the coordination functions necessary to handle communications between replicated application entities (*replicas*) at the multipoint communication session layer.

The tested portion of the IRp corresponds to the *decision making module* that corresponds to 3500 lines of C language. Figure 1 shows the target-system architecture for the fault-injection experiments. Three stations are considered that are connected via a Local Area Network.

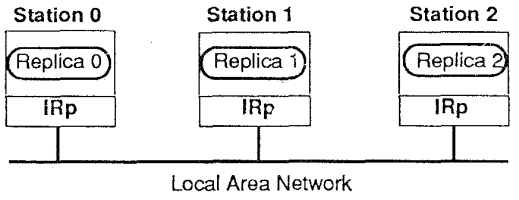

Figure 1. Delta-4 Target-System Architecture

An interactive simulator (about 21k C-language lines) developed to assist the designer when debugging the IRp program was made available by BULL France. In particular, the simulator provides the proper environment for the real IRp program to be executed concurrently on stations 0, 1, 2. The simulator was extensively used and validated prior to these experiments.

5.2 Derivation of the Execution Trees

The formalism $\langle e, r, a, d \rangle$ was used for generating the ET from the state transition model of the IRp. From this model, a set of assertions was derived as formal specification of the IRp. Set a, consisting of 157 assertions, has been derived from the state table of the protocol, provided by the designer. Set a is appreciably reduced to 54 by an interpreter of assertions that is written in C. Classes of equivalence (different edges in ET leading to the same action that activates the same post conditions) were found in a to reduce the complexity of the formal specification.

A set of WDF was obtained from ET and used as a test case for the IRp program. ET was used for designating a subset of injected faults in $\boldsymbol{F}$ that covered proper paths in the ET.

Fragments of the ET of stations 0, 1, 2, sensitized by the set of 8 first experiments that were performed manually on the 3-station architecture, are shown in figures 2a, 2b, 2c, respectively. The experiments have been identified by numbers at the leaves of the ET. However, several experiments can reach the same leaf. For example, experiments 1, 2, 4, 5 reach the same leaf (labeled 1, 2 and 4, 5, respectively) in figure 2a. The fact that each experiment reaches a leaf means that

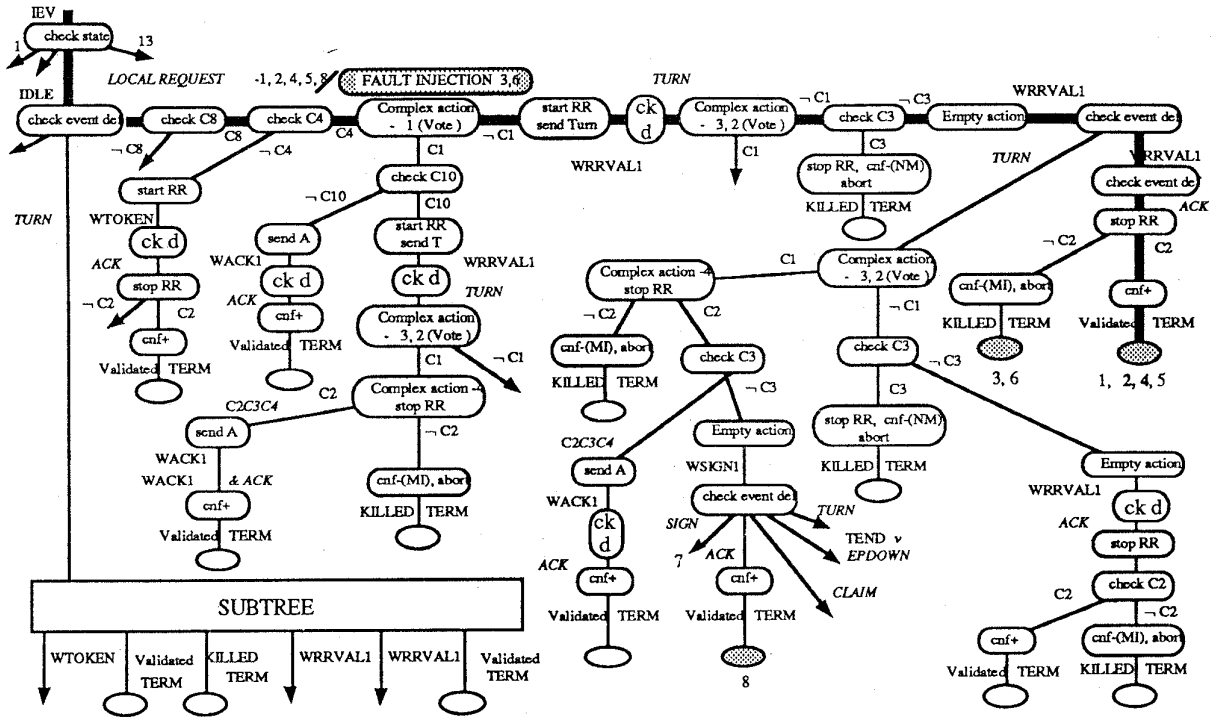

Figure 2a. Part of ET for IRp of Station 0

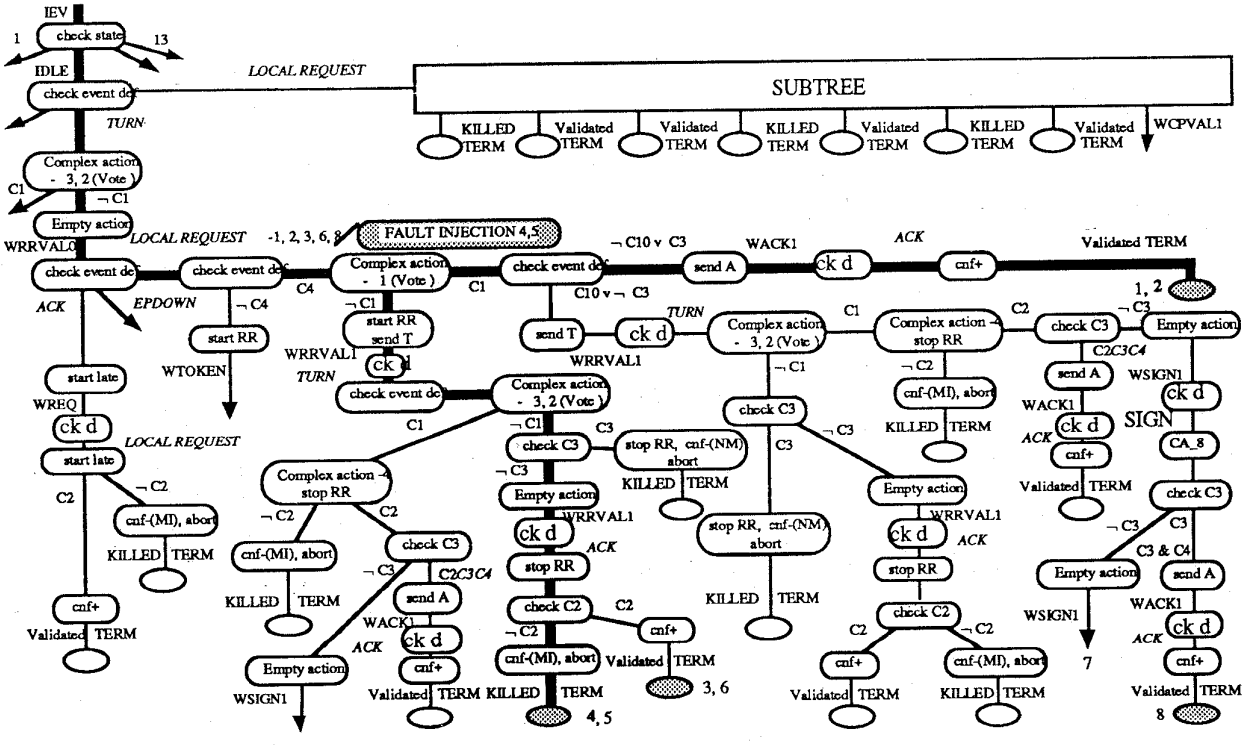

Figure 2b. Part of ET for IRp of Station 1

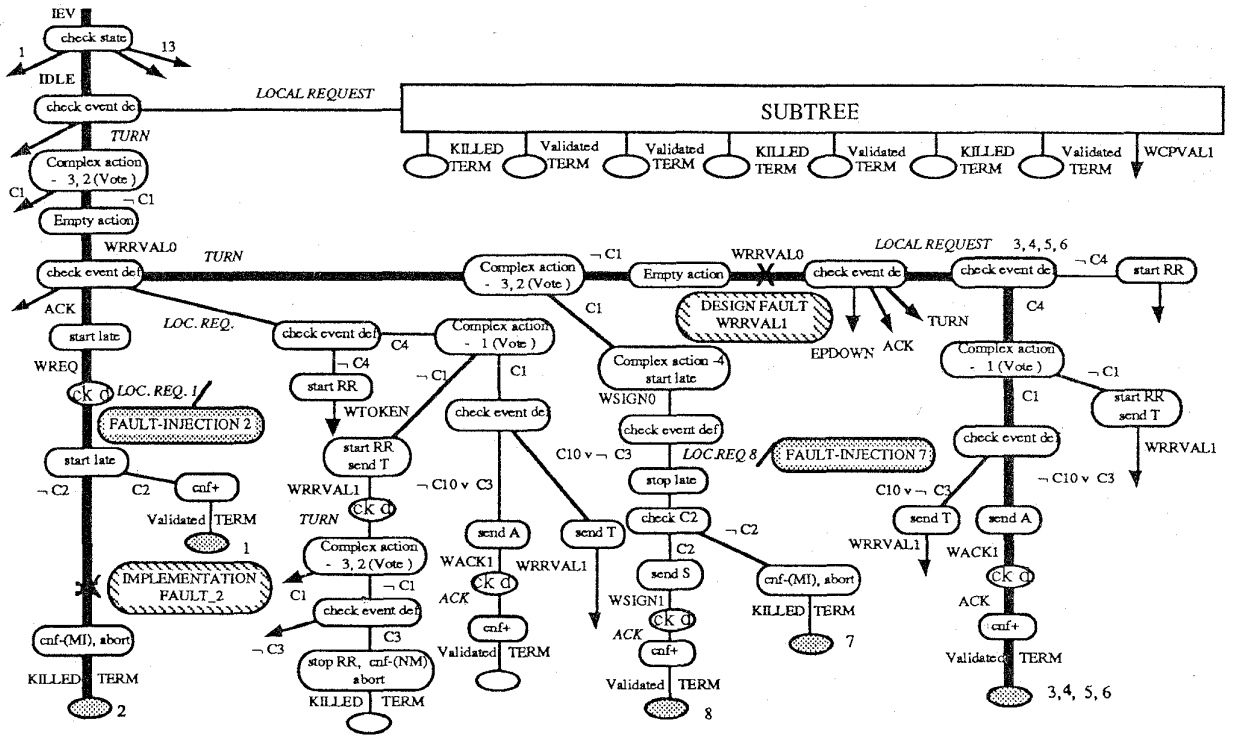

Figure 2c. Part of ET for IRp of Station 2]

- the appropriate values of the set of branching variables and event variables corresponding to the external events have been supplied; and
- the right path has been activated in all cases.

The faults (FAULT INJECTION j) are injected as external events (LOCAL REQUEST) containing wrong information. There exist 3 types of actions in these 8 experiments that cause the IRp to reach the event variable TERM; *ie*, the ET terminates:

1. cnf+: which leads to a VALIDATED-TERM;
2. cnf-(MI), abort: the local signature is not correct;
3. stop RR, cnf(NM), abort: no majority has been reached.

In cases 2 & 3, a KILLED-TERM is reached.

5.3 Revealing Design/Implementation Faults in the FTAM

We illustrate the fault-revealing power of FT/TS by considering in detail 2 fault-injection experiments. These 2 experiments (#2 & #4) have been selected because they illustrate both the experimental process and the type of analyses involved for revealing ftd-faults. For each experiment, IRp begins on each station with activated event variable IDLE. Experiment #2 illustrates the concurrent-activation process (table 3) for obtaining the Activated Execution Trees. The selection of the set of injected faults is based on lemma 1. The place & moment of the injection of the faults are determined from the concurrent static analysis of the ET (figures 2a, 2b, 2c). The number of injected faults is determined by the number of paths from the ET, which terminate with KILLED-TERM.

5.3.1 Experiment #2

It is anticipated that the activated ET terminate in states that set-up the 3 following EV (VALIDATED-TERM, VALIDATED-TERM, KILLED-TERM), for stations 0, 1, 2, respectively. Table 3 shows the subset of assertions describing this part of the protocol. The "¬" (NOT) is replaced by "!" in the current version of the automated tool VERIFY. The protocol is started by invoking LOCAL REQUEST on station 0. This activates internal EV WRRVAL1 and sends the message TURN to all stations.

When receiving this TURN, station 0 activates the next internal EV WRRVAL1 along the identified path. When receiving TURN, stations 1 & 2 activate internal EV WRRVAL0.

When external EV LOCAL REQUEST is activated, station 1 is directed to action SEND ACK to all stations.

When receiving this ACK, both stations 1 & 0 reach their required leaves (labeled 2). Reception of the ACK directs station 2 to activate EV WREQ. A fault is injected by simulating an erroneous LOCAL REQUEST (FAULT INJECTION 2). An anticipated behavior then is for the station to reach the leaf labeled 2 along the identified path.

The following paths are activated on different stations 0, 1, 2 after performing all these actions (see table 3):

- On station #0

LOCAL REQUEST & IDLE & !C1 ⇒ START RR, SEND T ⇒ TURN & WRRVAL1 & !C1 & !C3 ⇒ EMPTY ⇒ ACK & WRRAVL1 & C2 ⇒ cnf+ ⇒ VALIDATED-TERM

Table 3. Assertions' Subset

Experiment 2 - *Station* 0

IP(ACK & WRRAVL1 & C2 ‖WTOKEN & ACK & C2 ‖WCPVAL1 & ACK & C2 ‖WSIGN1 & ACK ‖WACK1 & ACK ‖WREQ & LOCAL REQUEST & C2 ‖ ⇒ cnf + ⇔ VALIDATED-TERM

IP(TURN & WRRVAL1 & !C1 & !C3 ‖ WRRVAL1 & CA-3 & !C3 & TEND ‖ WRRVAL1 & CA-3 & !C3 & EPDOWN ‖) ⇒ EMPTY ⇔ WRRVAL1

IP(LOCAL REQUEST & IDLE & !C1 ‖ LOCAL REQUEST & IDLE & C10 ‖ LOCAL REQUEST & WRRVALO & C4 & !C1 ‖) ⇒ START RR, SEND T ⇔ WRRVAL1

Experiment 2 - *Station* 1

IP(ACK & WRRAVL1 & C2 ‖ WTOKEN & ACK & C2 ‖ WCPVAL1 & ACK & C2 ‖ WSIGN1 & ACK ‖
ACK & WACK1 ‖ WREQ & LOCAL REQUEST & C2 ‖) ⇒ cnf + ⇔ VALIDATED-TERM

IP(LOCAL REQUEST & IDLE & C1 &!C10 ‖ TURN & WRRVAL1 & C1 & C2 & C3 & C4 ‖ WTOKEN & TURN & C4 & A-RR & !C10 ‖ WTOKEN & TURN & C4 & A-RR & C3 ‖ WTOKEN & CLAIM & C1 & C2 & !C10 & C4 ‖ WTOKEN & A-RR & !C10 & TEND ‖ WTOKEN & A-RR & !C10 & EPDOWN ‖ WTOKEN & A-RR & C3 & EPDOWN ‖ WTOKEN & A-RR & C3 & TEND ‖ LOCAL REQUEST & WRRVAL0 & C4 & C1 & !C10 ‖ LOCAL REQUEST & WRRVALO & C4 & C1 & C3 ‖ WCPVAL1 & CLAIM & C1 & C2 & !C10 & C4 ‖ WCPVAL1 & CLAIM & C1 & C2 & !C3 & C4 ‖ WSIGN1 & TURN & CA-8 & C3 & C4 ‖ WSIGN1 & CLAIM & CA-8 & C3 & C4 ‖ WSIGN1 & SIGN & CA-8 & C3 & C4 ‖ WSIGN1 & TEND & CA-6 & C7 & CA-9 & C3 & C4 ‖ WSIGN1 & EPDOWN & CA-6 & C7 & CA-9 & C3 & C4 ‖ WACK0 & TEND & CA-6 & C7 & CA-9 & C4 ‖ WACK0 & EPDOWN & CA-6 & C7 & CA-9 & C4 ‖) ⇒ SEND A ⇔ WACK1

IP(TURN & IDLE & !C1 ‖ TURN & WRRVAL0 & !C1 ‖ WRRVAL0 & TEND & CA-6 & C7 & CA-3 & !C9 ‖ WRRVAL0 & EPDOWN & CA-6 & C7 & CA-3 & !C9 ‖) ⇒ EMPTY ⇔ WRRVAL0

Experiment 2 - *Station* 2

IP(ACK & WRRAVL1 & C2 ‖ WTOKEN & ACK & C2 ‖ WCPVAL1 & ACK & C2 ‖ WSIGN1 & ACK ‖ ACK & WACK1 ‖ LOCAL REQUEST & WREQ & ! C2 ‖) ⇒ cnf + ⇔ KILLED-TERM

IP(ACK & IDLE ‖ ACK & WRRVAL0 ‖) ⇒ START LATE ⇔ WREQ

IP(TURN & IDLE & !C1 ‖ TURN & WRRVAL0 & !C1 ‖ WRRVAL0 & TEND & CA-6 & C7 & CA-3 & !C9 ‖ WRRVAL0 & EPDOWN & CA-6 & C7 & CA-3 & !C9 ‖) ⇒ EMPTY ⇔ WRRVAL0

- On station #1

TURN & IDLE & !C1 ⇒ EMPTY ⇒ LOCAL REQUEST & WRRVAL0 & C4 & C1 & !C10 ⇒ SEND A ⇒ ACK & WACK1 ⇒ cnf+ ⇒ VALIDATED-TERM

- On station #2

TURN & IDLE & !C1 ⇒ EMPTY ⇒ ACK & WRRVAL0 ⇒ START LATE ⇒ LOCAL REQUEST & WREQ &! C2 ⇒ cnf+ ⇒ KILLED-TERM ◄

However, when executing the actual program, the leaf "KILLED-TERM" could not be reached. This case can be classified as an *implementation fault*, according to the syndrome analysis in table 2. This fault corresponds to a fault made in the procedure to implement the validation of the signatures exchanged by the replicated entities and to control the branching variable C2. Intermittent erroneous behaviors of the implemented IRp for some activations of the program, induced by this fault, were known by the designer; however, it was not easy to diagnose precisely. The support provided by the testing strategy, the syndrome obtained, and the knowledge of the conditions for activating the paths in the activated trees were useful for diagnosing the fault. Accordingly, this fault could be easily corrected by the designer of IRp. Thus, the effectiveness of FT/TS for revealing implementation faults in the program implementation of the IRP was demonstrated.

5.3.2 Experiment #4

The activated ET should terminate in states that set-up the 3 following EV (VALIDATED-TERM, KILLED-TERM, VALIDATED-TERM), for stations 0, 1, 2, respectively. The EV KILLED-TERM is the predicted result associated with the fault injection labeled 4 on station 1.

The external event, which must occur on station 1 (figure 2b) for the proper path to be initiated, is the fault injection labeled 4 at the instant when the protocol activates EV WRRVAL0, for which:

- the Activation-Chain is:

IEV ⇒ CHECK STATE ⇒ IDLE ⇒ CHECK EVENT DEF ⇒ TURN ⇒ COMPLEX ACTIONS-3,2 (vote) ⇒ ¬C1 ⇒ EMPTY ACTION ⇒ WRRVAL0;

- the Activated Well-Defined Formula is:

Fault injection 4 ⇒ CHECK EVENT DEF ⇒ C4 ⇒ COMPLEX ACTION-1(vote) ⇒ ¬C1 ⇒ START RR, SEND T ⇒ WRRVAL1 ⇒ CHECK EVENT DEF ⇒ TURN ⇒ COMPLEX ACTIONS-3,2 (vote) ⇒ ¬C1 ⇒ CHECK C3 ⇒ ¬C3 ⇒ EMPTY ACTION ⇒ WRRVAL1 ⇒ CHECK EVENT DEF ⇒ ACK ⇒ STOP RR ⇒ CHECK C2 ⇒ ¬C2 ⇒ Cnf-(MI), ABORT ⇒ KILLED-TERM 4. ◄

The proper values of the set of branching variables (¬C1, ¬C2, ¬C3, C4,) and of the external EV, TURN, ACK are supplied by stations 0 & 2 in the same manner as explained for experiment #2.

An external event (labeled LOCAL-REQUEST-4) occurs in state IDLE of station 0 (figure 2a) to activate the proper path belonging to the set of predicted paths $\text{Pred}P_T(f_4)$, where:

- the A-Chain is:

IEV ⇒ CHECK STATE ⇒ IDLE;

- the Activated Well-Defined Formula is:

LOCAL REQUEST 4 ⇒ CHECK C8 ⇒ C8 ⇒ CHECK C4 ⇒ C4 ⇒ COMPLEX ACTION-1(vote) ⇒ ¬C1 ⇒ ... ⇒ Cnf+ ⇒ VALIDATED-TERM 4.

Respectively, the external event LOCAL REQUEST-4 is applied to state WRRVAL0 of station 2 (figure 2c) to activate

Table 4. Simulation Script for Experiment #4

Launch simulator	[tsf7 8(sim)] more exp4.s		
Initialize IRp	<cn3 elsaRR 012.s		
	Station 0	Station 1	Station 2
Code of the comm./ scheduling mode	9	9	9
Code of the station	0	1	2
Code selection	S_ DATA	S_DATA	S_DATA
MSAP name	30031	30031	30031
CNX name	3	3	3
Synchronization (Y/N)	*N*	*N*	*N*
Transfer type	S_MULTICAST	S_MULTICAST	S_MULTICAST
Transfer flags	S_METOO	S_METOO	S_METOO
Data	1	2	1
Type return to continue	⟨Return⟩	⟨Return⟩	⟨Return⟩
Experiment end [Station number (to quit >3)]	4		

the proper path belonging to the set of predicted paths $\text{Pred}P_T(f_4)$, where:

- the A-Chain is:

IEV ⇒ CHECK STATE ⇒ IDLE ⇒ CHECK EVENT DEF ⇒ TURN ⇒ COMPLEX ACTIONS-3,2(vote) ⇒ ¬C1 ⇒ EMPTY ACTION ⇒ WRRVAL0 ⇒ CHECK EVENT DEF ⇒ TURN ⇒ COMPLEX ACTIONS 3,2(vote) ⇒ ¬C1 ⇒ EMPTY ACTION ⇒ WRRVAL0;

- the Activated Well-Defined Formula is:

LOCAL REQUEST-4 ⇒ CHECK C4 ⇒ C4 ⇒ COMPLEX ACTION-1(vote) ⇒ C1 ⇒ CHECK EVENT DEF ⇒ (¬C10 v C3) ⇒ SEND ACK ⇒ WACK1 ⇒ CHECK EVENT DEF ⇒ ACK ⇒ Cnf+ ⇒ VALIDATED-TERM 4.

During generation of the ET of the IRp that corresponds to the behavior of station 2, the protocol did not terminate. It was halted at the edge labeled DESIGN FAULT WRRVAL1. According to the syndrome analysis depicted in table 2, we can classify this situation as an *eliminated design fault*, because the program did terminate when subjected to the test case defined by this experiment. This *eliminated design fault* corresponds to the use of a wrong variable (WRRVAL1 instead of WRRVAL0) in the set of assertions (state table); this fault subsequently was corrected in the implementation of the IRp, but remained in the state table that we used to obtain the set of assertions. FT/TS might have revealed such a fault as a design fault had the ET been fully constructed.

5.4 Design & Realization of a Test Sequence

The set of assertions and the derived activated ET (such as those in figure 2) provide a useful framework for specifying experiments that selectively exercise the program implementation of the IRp.

As an example, consider the process of making active a path in experiment #4 that terminates in a state, which sets up the Event Variable KILLED-TERM; the path has a leaf that is labeled with 4,5 in figure 2b. The respective A-Chain and A-WDF of the activated path are depicted in section 5.3.2. One can identify the following external event variables TURN, *fault injection* 4, TURN and ACK, which have to be provided in sequence to make the considered path active. The ET also identifies the conditions for synchronizing the injection of the fault with the state of station 1.

The interactive simulation environment developed by BULL to test communication protocols functionally was successfully used to support FT/TS when testing the FTAM within the IRp. Essentially, the simulator was used manually to perform the first set of fault injection experiments, such as those described in section 5.3. The main action during an experiment is to provide all necessary conditions (external EV, branching variables) for making active a selected ET path. Accordingly, the operator must provide data to the simulator when prompted to do so. These inputs are based on the information derived from the static analysis of the ET, as described above.

Table 4 depicts a typical form of the dialogue between the operator and the simulator to specify an experiment (more precisely, it corresponds to the specification of experiment #4). The inputs that are listed in each column for each station are entered by the operator in sequence, one station after another.

At the beginning, all three stations 0, 1, 2 (1001,1003, 1007), must be connected and MSAP(Multicast Service Access Point), S SAP (Session Service Access Point), SMAP(System Management Application Process), connection name (CNX name) need to be established. Such an activation is performed by a specific file (<cn3 elsaRR 012.s 9) shown in table 4.

Considering again experiment #4, the following steps must be performed:

1. The message TURN is to be sent by station 0. The A-Chains and AWD formulas for stations 0 & 2 are shown in section 5.3 and in figures 2a & 2c, respectively. Thus, the operator has to-be first on station 0. Station 0 has to send the message TURN to all stations (0 & 2) connected to MSAP-30031. To do this, the operator must: 1) provide the series of inputs shown in column 'Station 0' in table 4, and 2) activate the external variable - Local Request (Data 1).

2. Then, according to the A-Chain in figure 2b, station 1 activates the EV WRRVAL0. At that point, a synchronization

between the injected fault (Fault Injection 4) and the proper state of station 1 can be realized. For that, the operator has to-be on station 1 and inject the fault by providing wrong data (*eg*, 2 instead of 1, as shown in table 4). Station 1 then has to send a message TURN to activate WRRVAL1.

3. As indicated by ET of figure 2c, after receiving the message TURN twice, station 2 activates the EV WRRVAL0. The operator has to-be on station 2 to enter external event Local Request 4. Then, station 2 has to send a message ACK and to activate Validated TERM. ◄

After receiving a message ACK, station 1 should terminate and make active the message KILLED TERM. After receiving ACK, station 0 must terminate and then activate Validated TERM.

Although essentially we used the interactive simulator manually during the first set of experiments, it proved to be a useful tool for implementing FT/TS for complex FTAM. In particular, the design & realization of additional fault-injection experiments can be greatly facilitated — and to a large extent mechanized — by the generation of simulation scripts, analogous to the one in table 4. The scripts can be automatically & efficiently executed by the simulation environment.

Such a mechanization would be highly desirable for extending the coverage of the actual paths of the IRp program implementation when a set of predicted paths has been selected in the IRp activated ET. In particular, this would be readily achievable when considering the inputs to be provided to the Data row in the script in table 4. Randomly (or statistically) selected data can be easily applied there.

6. SUMMARY OF FT/TS

FT/TS is a *formal testing* approach for FT algorithms & mechanisms. To highlight the major features of FT/TS, figure 3 depicts a general framework that helps to identify the main steps of the testing strategy.

The left part of the diagram corresponds to the design & implementation steps. From the designer's intention (informal requirements), a formal model is established (*eg*, under the form of a state table, or flow graph). As shown by dashed arrows, another possibility is to present directly the behavior by a set of assertions. In this approach, the assertions are not generated from the code of the program; the model is used to elaborate the program. For testing purposes, a simulator is generally available for providing a proper environment for debugging the program. If the FTAM is a FT protocol, the simulator must enable the concurrent execution of the protocol program by analyzing a simulation script to provide necessary external conditions for exercising the program with several replicas of the application processes. Such concurrent execution enables one to test several paths simultaneously.

The r.h.s. of the graph describes the main steps of FT/TS. The main goal is to help the designer when performing a systematic test of the FTAM. Step #1 corresponds to defining a set of assertions characterizing the service anticipated from the program using FT/TS (see section 3). Although these assertions can be obtained directly from the designer's intention, they can also be derived from the model. An interpreter can provide possibilities for reducing the complexity of the formal specification. A tool can be used for constructing the tree of all possible executions from the set of assertions. The derivation of the ET includes testing for both missing or several activated assertions, which enables the set of assertions to be corrected (*viz*, removal of a *design fault*); such a feedback can also affect the model. Furthermore, well-defined formulas expressing specific service characteristics can be expressed within the formalism, and processed by the interpreter to verify their validity on the ET. To represent the behavior of a FT protocol for several replicated entities, several activated ET are considered for which distinct paths can be activated. The intent of this approach is to enforce the role of static analysis of the ET in the elaboration of *deterministic fault injection test sequences* for the program.

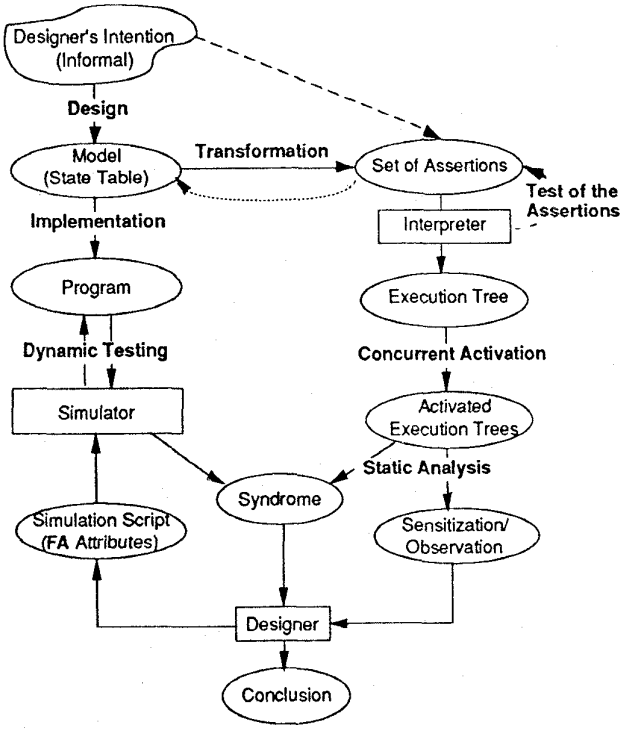

Figure 3. General Framework

In practice, such a goal can only be achieved under some constraints. In particular, controllability and/or observability restrictions do not always provide all the required conditions for the sensitization of a single path. Rather, the practically available conditions specify a *set of predicted paths* on the program (see section 4).

Static analysis performed with these activated trees provides two main results:

- the conditions of activation,
- the anticipated outcomes for the activated paths.

Both results characterize the sensitization & observation domains that guide the designer when elaborating on a systematic test case and when providing the fault (F) and activation (A) attributes to be incorporated into the simulation script to perform the fault injection experiment using the simulator.

The simultaneous dynamic testing of the program and of the concurrent activation of specific paths in the ET results in a syndrome that:

- characterizes the outcome of the test conducted for each side of the diagram,
- facilitates the diagnosis of fault tolerance deficiencies-faults (see table 2).

7. DISCUSSION

FT/TS was used successfully to conduct test sequences aimed at explicitly removing design/implementation faults in complex FT algorithms & mechanisms.

The main strength of FT/TS is its ability to test the FT algorithms & mechanisms in a systematic way by activating all paths of the associated ET. This systematic process enforces the power of fault injection to reveal potential FT deficiency faults. In particular, the strategy provides a framework for generating a functional deterministic test for programs implementing complex FT protocols.

The application of the strategy to a real protocol that supports the FT features of the Delta-4 distributed system illustrates its usefulness and its practical implementation. Even though the protocol had been independently verified formally, the application of the strategy revealed two FT deficiency-faults.

FT/TS is a heavy-weight approach, but is definitely needed to provide a high level of assurance when implementing FT algorithms & mechanisms. Indeed FT/TS is well suited to discover test-resistant faults, such as those revealed in the tested protocol. If the entire ET has been built, FT/TS can guarantee the absence of faults in the FT algorithms & mechanisms.

The study reported here describes an early (mostly manual) application of FT/TS for which only a limited number of experiments were conducted. However, the formalism permits the mechanization of major steps of the strategy. In particular, it is possible to use an interpreter of assertions to reduce automatically the set of assertions by identifying classes of equivalence. Other possible mechanizations concerning the construction of the complete ET, running a sequence of experiments, are being implemented. Accordingly, a tool (VERIFY) has been written to support the application of FT/TS for the formal verification of software. This is written in C++ and currently has about 6k lines. VERIFY has been implemented on SPARC workstations running the Solaris 2.4 operating system. It is an interactive tool and has been successfully used to derive the 57 compressed assertions that fully describe the Delta-4 protocol and the set of all possible paths (27 paths). However, no further ftd-faults were unveiled as a result of constructing all execution paths. Further applications of VERIFY concern two additional major protocols for NASA & AT&T. The communication multimedia protocol for AT&T Bell Laboratories has been presented by 131 assertions. As well, a state-explosion problem was not observed during the generation of the ET. In this case, the set contains 35 paths and the average number of levels in the ET is 30. No state-explosion problem was observed by generating all possible paths for any of these protocols.

ACKNOWLEDGMENT

We are pleased to thank David Powell and Pascale Thévenod (from LAAS) for pertinent discussions during the early stages of this research, and Philippe Reynier (from BULL) for his help with the simulation environment. This work was performed within the framework of PDCS, ESPRIT Basic Research Action #3092, "Predictably Dependable Computing Systems".

REFERENCES

[1] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology", *Proc. 15th Int'l Symp. Fault-Tolerant Computing* (FTCS-15), 1985, pp 2-11; IEEE Computer Society Press.

[2] J.-C. Laprie (Ed), *Dependability: Basic Concepts and Terminology in English, French, German, Italian, and Japanese,* 1992; Springer-Verlag.

[3] J. Arlat, Y. Crouzet, J.-C. Laprie, "Fault injection for the experimental validation of fault tolerance", *Proc. Esprit Conference (Esprit'91)* (CEC-DGXIII), 1991, pp 791-805; Office of Official Publications of the European Communities.

[4] Z. Segall, D. Vrsalovic, D. Siewiorek, *et al*, "FIAT % Fault injection-based automated testing environment", *Proc. 18th Int'l Symp. Fault-Tolerant Computing* (FTCS-18), 1988, pp 102-107; IEEE Computer Society Press.

[5] U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", *Proc. 19th Int'l Symp. Fault-Tolerant Computing* (FTCS-19), 1989, pp 340-347; IEEE Computer Society Press.

[6] R. Chillarege, N.S. Bowen, "Understanding large system failures % A fault injection experiment", *Proc. 19th Int'l Symp. Fault-Tolerant Computing* (FTCS-19), 1989, pp 356-363; IEEE Computer Society Press.

[7] J. Arlat, M. Aguera, L. Amat, *et al*, "Fault injection for dependability validation % A methodology and some applications", *IEEE Trans. Software Engineering*, vol 16, 1990 Feb, pp 166-182.

[8] C.J. Walter, "Evaluation and design of an ultra-reliable distributed architecture for fault tolerance", *IEEE Trans. Reliability*, vol 39, 1990 Oct, pp 492-499.

[9] G.S. Choi, R.K. Iyer, "FOCUS: An experimental environment for fault sensitivity analysis", *IEEE Trans. Computers*, vol 41, 1992 Dec, pp 1515-1526.

[10] K.K. Goswami, R.K. Iyer, "Simulation of software behavior under hardware faults", *Proc. 23rd Int'l Conf. Fault-Tolerant Computing* (FTCS-23), 1993, pp 218-227; IEEE Computer Society Press.

[11] J. Arlat, M. Aguera, Y. Crouzet, *et al*, "Experimental evaluation of the fault tolerance of an atomic multicast protocol", *IEEE Trans. Reliability*, vol 39, 1990 Oct, pp 455-467.

[12] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, "Using heavy-ion radiation to validate fault-handling mechanisms", *IEEE Micro*, vol 14, 1994 Feb, pp 8-23.

[13] J.H. Barton, E.W. Czeck, Z.Z. Segall, D.P. Siewiorek, "Fault injection experiments using FIAT", *IEEE Trans. Computers*, vol 39, 1990 Apr, pp 575-582.

[14] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, "FERRARI: A flexible software-based fault and error injection system", *IEEE Trans. Computers*, vol 44, 1995 Feb, pp 248-260.

[15] D.R. Avresky, P.K. Tapadiya, "Software-based fault-injection tool for validation of software fault-tolerant techniques under hardware faults", *Second ISSAT Int'l Conf. Reliability & Quality in Design*, 1995 Mar.

[16] E. Jenn, J. Arlat, M. Rimén, *et al*, "Fault injection into VHDL models: The MEFISTO tool", *Proc. 24th Int'l Symp. Fault-Tolerant Computing* (FTCS-24), 1994, pp 66-75; IEEE Computer Society Press.

[17] J. Arlat, "Fault injection for the experimental validation of fault-tolerant systems", *Proc. Workshop Fault-Tolerant Systems,* 1992, pp 33-40; IEICE.

[18] J.A. Clark, D.K. Pradhan, "Fault injection % A method for validating computer-system dependability", *Computer*, vol 28, 1995 Jun, pp 47-56.

[19] T. Dilenno, D. Yaskin, J.H. Barton, "Fault tolerance testing in the advanced automation system", *Proc. 21st Int'l Symp. Fault-Tolerant Computing* (FTCS-21), 1991, pp 18-25; IEEE Computer Society Press.

[20] K. Echtle, Y. Chen, "Evaluation of deterministic fault injection for fault-tolerant protocol testing", *Proc. 21st Int'l Symp. Fault-Tolerant Computing* (FTCS-21), 1991, pp 418-425; IEEE Computer Society Press.

[21] K. Echtle, M. Leu, "The EFA fault injector for fault-tolerant distributed system testing", *Proc. Workshop Fault Tolerant Parallel & Distributed Systems*, 1992, pp 28-35; IEEE Computer Society Press.

[22] D. Avresky, J. Arlat, J.-C. Laprie, Y. Crouzet, "Fault injection for the formal testing of fault tolerance", *Proc. 22nd Int'l Symp. Fault-Tolerant Computing* (FTCS-22), 1992, pp 345-354; IEEE Computer Society Press.

[23] D. Powell (Ed) *Delta-4: A Generic Architecture for Dependable Distributed Computing*, 1991; Springer-Verlag.

[24] M. Diaz, C. Vissers, "SEDOS: Designing open distributed systems", *IEEE Software*, vol 16, 1989 Aug, pp 24-33.

[25] A. Citimile, U. Carlini, "Reverse engineering: Algorithms for program graph production", *Software Practice & Experience*, vol 21, 1991 May, pp 519-537.

[26] D. Brand, W.H. Joyner, "Verification of protocols using symbolic execution", *Computer Networks*, 1978 Feb.

[27] B. Korel, "Automated software test data", *IEEE Trans. Software Engineering*, vol 16, 1990 Aug, pp 870-879.

[28] J.-C. Fernandez, J.-L. Richier, J. Voiron, "Verification of protocol specifications using the Cesar system", *Protocol Specification, Testing, Verification* (M. Diaz, Ed) vol V, 1986, pp 71-90; North-Holland.

[29] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, "An experimental study of software structural testing: Deterministic versus random input generation", *Proc. 21st Int'l Symp. Fault-Tolerant Computing* (FTCS-21), 1991, pp 410-417; IEEE Computer Society Press.

[30] M. Chérèque, D. Powell, P. Reynier, *et al*, "Active replication in Delta-4", *Proc. 22nd Int'l Conf. Fault-Tolerant Computing Systems* (FTCS-22), 1992, pp 28-37; IEEE Computer Society Press.

AUTHORS

Dr. D. R. Avresky; Dep't of Computer Science; Texas A&M Univ; College Station, Texas 77843-3112 USA.
Internet (e-mail): avresky@cs.tamu.edu

Dimiter R. Avresky (M) is with the Department of Computer Science at Texas A&M University; this work was done while he was with LAAS-CNRS in Toulouse. His research interests focus on hardware & software fault-tolerant systems, parallel computers, functional languages, parallel programming, testing, and fault-injection. He is an editor and co-author of *Hardware and Software Fault-Tolerance in Parallel Computing Systems*, 1992, and *Fault-Tolerant Parallel and Distributed Systems*, 1995. He is Member of IEEE and The New York Academy of Sciences.

Dr. Jean Arlat; LAAS-CNRS; 7, Ave du Colonel Roche; 31077 Toulouse cedex, FRANCE.
Internet (e-mail): arlat@laas.fr

Jean Arlat is Directeur de Recherche of CNRS, the French National Organization for Scientific Research. He joined LAAS-CNRS in 1976. His research has focused on dependable computing and, more specifically, on analytic & experimental validation of fault-tolerant computers. He received the Certified Engineer Degree (1976) from the National Institute of Applied Sciences, Toulouse; the Doctor in Engineering Degree (1979) in Automatic Control and the Doctor of Sciences Degree (1990) in Computer Science from the National Polytechnic Institute of Toulouse. In 1994-1995, he was Chair'n of the IEEE Computer Society TC on Fault-Tolerant Computing.

Dr. Jean-Claude Laprie; LAAS-CNRS; 7, Ave du Colonel Roche; 31077 Toulouse cedex, FRANCE.
Internet (e-mail): laprie@laas.fr

Jean-Claude Laprie is Directeur de Recherche of CNRS. He joined LAAS-CNRS in 1968, where he has directed the research group on Fault Tolerance and Dependable Computing since 1975. His research has focused on dependable computing since 1973, and especially on fault-tolerance and dependability evaluation. He received the Certified Engineer Degree (1968) from the Higher National School for Aeronautical Constructions, Toulouse; the Doctor in Engineering Degree (1971) in Automatic Control and the Doctor of Sciences Degree (1975) in Computer Science from the University of Toulouse. In 1984-1985, he was Chair'n of the IEEE Computer Society TC on Fault-Tolerant Computing, and was the Chair'n of IFIP WG10.4 on Dependable Computing and Fault Tolerance from 1986 to 1995.

Dr. Yves Crouzet; LAAS-CNRS; 7, Ave du Colonel Roche; 31077 Toulouse cedex, FRANCE.
Internet (e-mail): crouzet@laas.fr

Yves Crouzet is Charg de Recherche of CNRS. He joined LAAS-CNRS in 1975. His research has focused on dependable computing and more specially on experimental validation of fault-tolerant computers and software testing approaches. He received the Certified Engineer Degree (1975) from the Higher National School of Electronics, Electrical Engineering, Computer Science and Hydraulics, Toulouse; the Doctor in Engineering Degree (1978) in Automatic Control and the Habilitation diriger des Recherches (1995) in Computer Science from the National Polytechnic Institute of Toulouse

Manuscript received 1995 August 14

Publisher Item Identifier S 0018-9529(96)07148-5 ◄TR►