Deriving test sets from partial proofs

Guillaume Lussier, Hélène Waeselynck

LAAS-CNRS
7 Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
E-mail: {glussier,waeselyn}@laas.fr

Abstract

Proof-guided testing is intended to enhance the test design with information extracted from the argument for correctness. The target application field is the verification of fault-tolerance algorithms where a complete formal proof is not available. Ideally, testing should be focused on the pending parts of the proof. The approach is experimentally assessed using the example of a group membership protocol (GMP), a complete proof of which has been developed by others in the PVS environment. In order to obtain a partial proof example, we proceed to flaw insertion into the PVS specification. Test selection criteria are then derived from the analysis of the reconstructed (now partial) proof. Their efficiency for revealing the flaw is experimentally assessed, yielding encouraging results.

1. Introduction

Functional testing approaches usually rely on coverage measures, test purposes, or selection hypotheses associated with models of behavior. Such criteria are used to select finite test sets from the models. They always involve assumptions. For example, transition coverage assumes that flaws manifest themselves as simple output or transfer errors. Test purposes represent pieces of behavior that are deemed important to be tested. Uniformity hypotheses are used to group inputs that should be equivalent in their capability of stimulating the system under test. In this paper, we investigate whether a partial formal proof can be a useful basis for deriving such assumptions.

The target application field is the verification of Fault-Tolerance (FT) algorithms. As FT mechanisms are critical components for building dependable architectures, strong evidence for correctness of the underlying algorithms is desirable. Suppose, however, that a complete formal proof could not be obtained. Then, testing can be seen as a complementary technique to gain confidence that the algorithm should be correct, or to exhibit counter-examples under the form of test scenarios. The tested artifact is possibly a prototype of the algorithm, or a specification that can (in some way) be executed. Ideally, the design of testing should take advantage of the fact that a proof has been attempted. For example, the test size can be reduced if the algorithm requirements have formally been proved to hold for a subset of the input space. An unsuccessful proof by cases might suggest test cases that would be potentially significant to the correctness of the algorithm. Intuitively, one would expect potential flaws to be somehow related to the pending parts of the proof.

While the idea of *proof-guided testing* seems appealing, its feasibility and efficiency have to be studied on realistic examples. We adopt an experimental approach: starting from incomplete proofs of flawed FT algorithms, we investigate whether the proof analysis does supply useful information for guiding the design of testing.

Our previous work along these lines [4, 5] addressed testing from *informal* proofs, that is, paper demonstrations done by usual reasoning. Our conclusion was that such proofs may carry relevant information for testing, but this depends on their degree of rigorousness. In [4], the analysis of the proof revealed major flaws of reasoning, and proof-guided testing was unsuccessful. The proof example studied in [5] was much better crafted than the previous one (but still flawed), and allowed us to identify an input subspace that yielded a high failure rate of the algorithm. In this paper, we are now considering the case of *formal*, but partial, proofs.

For experimental purposes, it was easy to find in the literature examples of incorrect FT algorithms "proved" by informal demonstration. However, examples of partial formal proofs for incorrect algorithms are more difficult to get, as it is only the successful proofs that are made available in the public domain. We decided to proceed as follows: obtain a successful proof, insert a flaw into the specification of the algorithm, and then use the accordingly modified – and now partial – formal proof as a case study for proof-guided testing.

Section 2 presents the background of the example studied in this paper. The FT algorithm is a Group Membership Protocol (GMP). Its formal proof [8] has been developed in the PVS [6] environment. Section 3 gives an overview of our experimental approach. Section 4 described the GMP algorithm, its requirements, and first analysis results from a testing perspective. After a general presentation of the original proof in Section 5, we proceed to flaw insertion in Section 6. Experimental test results for the flawed algorithm are given in Section 7.

2. Background of the Case Study

In a distributed system, a group membership service allows non-faulty processors to agree on their membership and to exclude faulty ones. The studied algorithm is the membership service offered by the *Time-Triggered Protocol* (TTP). TTP [3] has been developed over the past twenty years at the university of Vienna, and is now commercially promoted by TTTech. It is an integrated communication protocol for time-triggered architectures, typically used for automotive functions (brake-by-wire, steer-by-wire), or avionics ones (the communication system of the Airbus A380 cabin pressure control system will be based on TTP).

The complexity of the behavior of the group membership protocol (GMP), and its tight interactions with other TTP services, makes it difficult to formally analyze it. Several attempts were necessary before a complete formal proof could be developed.

A related GMP algorithm, proposed in [2], was first proved by detailed but informal demonstration. The authors used model-checking of an instance of the algorithm to consolidate their paper demonstration for the generic case. Unfortunately, the protocol was found flawed after publication[1]. This experience led one of the authors (John Rushby) to formally rework the problem, using the PVS verification system. He eventually succeeded in doing this, but had to develop an original proof method, presented in [11].

This proof method has been later reused at the University of Ulm to prove the TTP GMP. As the proof was progressing, the protocol and its PVS formalization went through successive versions [7, 8, 10]. Our experimental study is based on the last version presented in [8], for which we could obtain the PVS source files and proof scripts.

3. Experimental Approach

Given a partial proof, the proposed approach to designing test sets involves three steps.

[1]Note that we used this knowingly flawed example to support previous investigation on testing from informal proofs [5].

- **High-level analysis.** The aim of the analysis is to gain an understanding of the FT algorithm and its requirements: under certain assumptions, some key properties are to be fulfilled. The assumptions include a model of the faults to be tolerated, as well as other environmental assumptions. From their identification, a definition of the algorithm's test input domain is derived. The key properties yield a specification of the test oracle checking acceptance or rejection of the test results. The understanding of the algorithm must be sufficient to initiate development of a prototype to be tested, in case the specification environment does not offer adequate support for submitting test sets to the formal model (we had to develop such a prototype for the GMP case study).

- **Detailed analysis.** The PVS source files are thoroughly analyzed, so as to gain deep insight into the proof structure. The aim is the identification of the pending parts of the proof, which will be used to direct testing in the next step of the approach. The proof analysis can be conducted at two levels. The first level considers a macroscopic view of the proof structure in terms of intermediate lemmas. It must be understood how pending lemmas contribute to the building of a global proof of the key properties. The second level refines the previous analysis by considering the proof trees associated with each pending lemma: analysis is then performed in terms of undischarged proof sequents in the trees. Our experiments will consider both levels of analysis. It is anticipated that analysis at the sequent level be more difficult than at the lemma level: in the framework of the case study, it will be investigated to what extent the more difficult analysis allows us to improve the effectiveness of testing.

- **Proof-guided testing.** This steps consists in exploiting the results of the previous analysis, whether at the lemma or sequent level, to guide the design of testing. The identified weaknesses of the partial proof are used to determine test selection criteria, i.e. to determine functional cases to be activated during testing. Then the generation of test sets is performed following a probabilistic approach, statistical testing [12]. Statistical testing aims to compensate for the imperfect connection of common test criteria with the flaws to be revealed: the cases identified by a criterion have to be exercised several times with different random test data. In this way, there is no need for a perfect match between identified cases and revealing inputs. In our experimental framework, we evaluate the efficiency of proof-guided testing in terms of induced failure rate of the algorithm (the higher the rate, the better the efficiency), and in terms of improvement with respect to a

blind sampling profile.

Since the studied GMP has been completely proved in the PVS environment, it should be correct with respect to its key properties – provided its formal specification is accurate, which is an important problem but falls outside the scope of this paper. Hence, for this case study, there is no proof weakness toward which testing should be directed. For experimentation purpose, we propose to insert a flaw into the specification of the algorithm and then study whether the accordingly modified – and now partial – formal proof may be helpful to guide the design of testing. In practice, detailed analysis is first performed on the original PVS specification: a fine understanding of the complete proof is necessary to be able to proceed to flaw insertion (see below). After flaw insertion, detailed analysis is focused on the resulting partial proof.

The process of **flaw insertion** is shown in Figure 1. The inserted flaws consist of modifications of the PVS description of the algorithm. Once such a modification has been introduced, a number of lemmas become unproved, or even ill-defined. Thus, the modification has to be propagated throughout the PVS model and its proof, which involves formal reworking. The extent of formal reworking may be more or less large, depending on the inserted flaw. Definitions and lemmas directly impacted by the algorithm's modification are first reworked. Proof tactics associated to these lemmas may also have to be adapted. Then, the modified lemmas may necessitate reworking of the general proof structure, yielding further modification. The process ends when the reworked partial proof is deemed representative of a genuine attempt to prove the modified algorithm. Note that, for practical reasons, we did not consider flaws necessitating major changes in the proof structure.

We now present the results of this experimental approach applied to the GMP example.

4. High-level analysis

4.1. Assumptions and key properties

The studied GMP involves *n* processors (numbered *0*, ..., *n-1*) attached to a broadcast bus. Execution is synchronous, with a global time *t* increased by one at each step. At time *t*, processor *t* mod *n* is the only one that can broadcast messages. This defines broadcast *slots* owned by this processor. Each processor maintains a local view of the membership set, i.e., the set of processors it considers non-faulty. Whenever its slot is reached, a processor will remain silent if it is no more contained in its own membership set (it has diagnosed itself as faulty). Otherwise, it sends a message including information on its local view of the membership. More precisely, it appends to the message a CRC checksum calculated over the message data and its membership set.

Only two types of faults are considered:

- Send faults. The broadcaster either fails to produce activity on the bus, or performs an incorrect sending of the message. Since broadcasts are assumed consistent, none of the non-faulty processors receives a correct message.
- Receive faults. The affected processor fails to receive a broadcast.

Once a processor has become faulty (first manifestation of a fault), it may or may not succeed in sending or receiving messages in the subsequent slots. Only one non-faulty processor can become faulty in any 2n consecutive steps, and there are always at least two non-faulty processors in the system.

Under these assumptions, the GMP has to fulfill three properties at any time:

- **Validity.** Non-faulty processors should have all the non-faulty processors in their membership sets, and at most one faulty processor in their sets (as it may take some time to diagnose the fault). Faulty processors should either have removed themselves from their sets, or have a subset of the non-faulty processors plus themselves in their sets.
- **Agreement.** All non-faulty processors should have the same membership sets.
- **Self-diagnosis.** A processor that becomes faulty should diagnose its fault and remove itself from its own membership set in less than 2n steps.

4.2. Presentation of the algorithm

A detailed explanation of the GMP behavior can be found in [8]. Here, we reproduce a description of the algorithm under the form of guarded commands ($guard \longrightarrow action$), and give a general outline of it.

Figure 2 presents the 14 guarded commands defining the behavior of a processor *p* at slot *t*, according to its mode at that slot (broadcaster, receiver). In receiving mode, the current broadcaster is processor *b*. The guards are evaluated in top-down order, and processor *p* executes the action corresponding to the first guard that evaluates to true. The membership set of *p* at time *t* is denoted mem_p^t.

In receiving mode, the arrival (or non arrival) of a message determines the following input variables:

- $arrives_p^t$ is a Boolean variable set to true if processor p correctly receives a message at step t.
- $null_p^t$ is a Boolean variable set to true if p did not detect any traffic on the bus at step t.
- mem_b^t is the *membership* set sent by b (when $arrives_p^t$ is true).

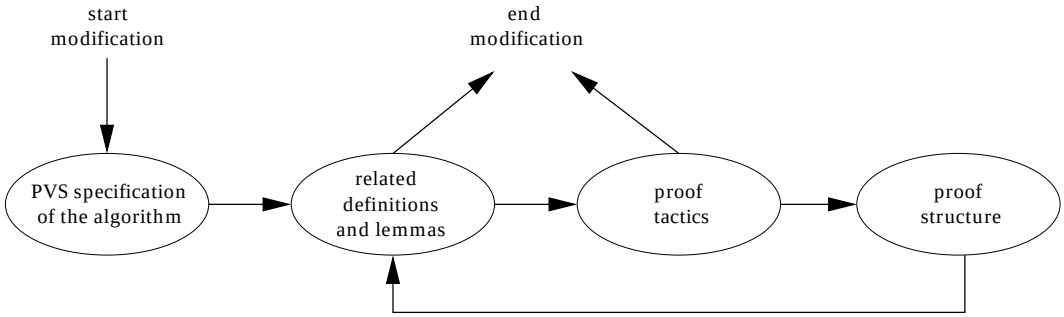

Figure 1. Flaw insertion in the PVS specification and proof

Broadcaster :

(1)	$acc_p^t > rej_p^t \wedge$ $acc_p^t \geq 2$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $prev_p^{t+1} = T \wedge$ $acc_p^{t+1} = 1 \wedge$ $rej_p^{t+1} = 0$
(2)	otherwise	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{p\}$

Receiver :

(3)	$p \notin mem_p^t$	$\longrightarrow$	no change
(4)	$prev_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{p, b\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $prev_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(5)	$prev_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{b\} \setminus \{p\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $prev_p^{t+1} = F \wedge$ $doubt_p^{t+1} = T \wedge$ $rej_p^{t+1} = rej_p^t + 1 \wedge$ $succ_p^{t+1} = b$
(6)	$prev_p^t \wedge null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(7)	$prev_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$
(8)	$doubt_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{p, b\} \setminus \{succ_p^t\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $doubt_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(9)	$doubt_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{succ_p^t, b\} \setminus \{p\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \cup \{succ_p^t\} \setminus \{p\} \wedge$ $doubt_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(10)	$doubt_p^t \wedge null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(11)	$doubt_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$
(12)	$arrives_p^t \wedge (mem_p^t = mem_b^t)$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(13)	$null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(14)	otherwise	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$

Figure 2. Guarded commands of the GMP (from [8] and the PVS source code)

Note that the description of the algorithm makes the conceptual assumption that a message contains the broadcaster's local view of the membership, while it actually contains a CRC checksum. This is legitimate because the receiver p can perform a CRC check on the received message data and its own membership view. If the two checksums are the same, p can conclude (with a certain probability) that the membership views are the same, as in guard (12). In case of mismatch, p cannot directly check the identity of processors about which p and b disagree. But p can try to reconstruct b's membership set by performing CRC calculations with certain entries of its own membership set changed, as in guards (4), (5), (8), (9).

The commands in receiving mode can be classified into four categories, depending on the internal variables appearing in their guard :

- command (3), guarded by p not belonging anymore to its own membership set. Its state is then frozen.
- commands (4) to (7), guarded by the $prev_p^t$ Boolean variable, capturing the fact that p considers it was the previous non-faulty broadcaster. $prev_p^t$ was set to true by command (1) in broadcasting mode, and is reset to false by commands (4) and (5).
- commands (8) to (11), guarded by the $doubt_p^t$ Boolean variable, true whenever p considers that it may have suffered a send fault during its previous broadcast. $doubt_p^t$ is set to true by command (5). It is reset to false when p is able to conclude that it did not suffer a send fault (command (8)), or that it did suffer one (command (9)).
- commands (12) to (14), corresponding to the standard case where p belongs to its own membership set, is not the previous broadcaster, and has no doubt on its previous broadcast.

Commands guarded by $prev_p^t$ and $doubt_p^t$ are consistent with an implicit acknowledgement mechanism, done by broadcasting membership information. Let us assume that p suffers a send fault at slot t. At $t+1$, it may correctly receive a message from processor b, and observe that it is no more included in b's membership set (guard (5)). It then concludes that b did not receive its message, but does not know whether it comes from b having suffered a receive fault, or from itself having suffered a send fault. By default, b is excluded from p's membership, but the $doubt$ variable of p is set to true. When a second message confirms that p's broadcast was incorrect, p will remove itself from its own membership set.

However, this mechanism is not sufficient to ensure self-diagnosis in any case. This is why p also uses two counters, acc and rej, representing the number of messages it has accepted or rejected since its previous broadcast. Generally speaking, acc is increased by one if p correctly receives a message and agrees with the membership view of the broadcaster. Counter rej is increased by one if p receives an incorrect message, or receives a correct one but disagrees with the broadcaster's membership view. The values of the counters are periodically checked, at each broadcast slot of p (guards (1) and (2)). The values allow p to diagnose its fault if it has rejected more messages than it has accepted since its last broadcast, or if it has accepted none (*acc* was reset to 1 at the slot of its previous broadcast).

4.3. Results of the high-level analysis

At this stage of high-level analysis, we are able to define the test input domain, and the test oracle checks. We are also able to initiate the development of a prototype of the algorithm.

The detailed description of the algorithm makes it straightforward to implement a GMP prototype. The C code we developed is a quasi-literal transcription of the PVS code corresponding to Figure 2.

The test oracle is specified to check the validity, agreement and self-diagnosis properties at each step (see Section 4.1). The implementation of the checks is closely based on the PVS representation of these invariant properties. It requires that all local membership sets be observed at each step.

The definition of the test input domain is less straightforward. The identification of the GMP assumptions (briefly presented in section 4.1), required a careful analysis of all axioms extracted from the PVS specification. We found that the $null_p^t$ inputs were under-specified. We decided to place restrictions on the situations allowed by the axioms. As an example, from the PVS axioms, nothing prevents $arrives_p^t$ and $null_p^t$ from being both true at the same time. This seems meaningless, as p cannot both receive a correct message[2] and detect no traffic on the bus. This situation was not allowed in our definition of the input domain. For non-faulty receivers, or for faulty ones not manifesting their fault at step t, input $null_p^t$ is true if and only if no traffic is generated on the bus. This may correspond to one of the following cases: (1) the broadcaster decides to remain silent (because it is no more in its own membership set), (2) the broadcaster manifests a fault and fails to send anything on the bus. For faulty receivers manifesting a fault at step t, $null_p^t$ may take any value true or false: whatever the broadcaster's behavior, the faulty receiver either receives nothing, or receives something that it cannot interpret as a correct message. Note that we do not exclude the situation where the faulty receiver wrongly detects activity on the bus while the broadcaster remains silent. This may be a debatable decision, but anyway the situation is allowed by the axioms.

[2]The axioms ensure that it is impossible to correctly receive a message that was not correctly sent

We chose to define a test input sequence by the number n of processors, with n>2 (at least two non-faulty processors), and by a list of faults affecting the system. A fault is characterized by its occurrence time, its type and the affected processor. Following the previous discussion, Figure 3 tabulates the four fault types that may affect a processor p in our test environment. The fault type must be consistent with the occurrence slot and affected processor (e.g., a send fault affects the broadcaster at that slot). Moreover, let us recall that there are constraints on the maximum number of faults (n-2) and the temporal dispersion of faults affecting processors non-faulty so far (2n slots apart).

At this stage, we were able to implement a crude random profile generating valid test sequences for systems from 3 to 20 processors, a range targeted by the Time-Triggered Architecture. The crude profile was implemented not only for experimental comparison with more designed profiles, but also for another reason: it allowed us to ensure that we were able to extract, from the PVS axioms, a constructive definition of the test input domain (under the form of a random generation function).

5. Detailed analysis of the GMP proof

5.1. The disjunctive invariants proof method

The GMP proof aims at establishing that the validity, agreement and self-diagnosis properties hold at any time. Usually, such invariant properties are verified by an induction proof. Since the properties to be proved are generally not inductive, they have first to be strengthened by conjoining additional properties, until an inductive invariant is obtained. This classical approach has been unsuccessful when applied to a related, and much simpler, GMP algorithm (see Section 2, mentioning this algorithm [2]). Proof attempts were defeated by the number and complexity of the auxiliary invariants, and by case explosion.

In [11], J. Rushby proposed a new method to tackle the problem. The principle is to strengthen the property of concern into a disjunction of "configurations" that can easily be proved to be inductive. The set of configurations, and transitions among configurations, have a diagrammatical representation that conveys insight into the operation of the algorithm.

The so-called *Disjunctive Invariants* method has been successfully used by H. Pfeifer from Ulm to prove two key services of the TTP: the GMP we use as a case study, and the clock synchronization service [9] (this justifies the global time assumption made for the GMP).

5.2. Configuration diagram for the GMP

The GMP proof is based on the *Configuration Diagram* reproduced in Figure 4. A formal description of this diagram is given in the PVS source files. Detailed comments about the construction and formalization of this diagram can be found in [8].

The diagram can be seen as a description of an abstract state machine, representing the operation of the GMP for an unbounded number n of processors. The vertices are configurations, defined by predicates and corresponding to properties of the global state of the system. The configurations are parameterized by the time t, and by variables denoting processors: x is the processor that most recently became faulty, z is the most recent non-faulty broadcaster, and b is the current broadcaster. Each transition corresponds to the execution of one step of the algorithm, and is conditioned by a predicate.

The diagram is iteratively developed by performing symbolic reachability analysis. One starts by defining an initial configuration (here, *stable*) including the initial states of all processors. Then, from an existing configuration, one invents some transition conditions for it. The conditions are not necessarily mutually disjoint, but their disjunction must be true. For each condition, one symbolically simulate a step of the algorithm by rewriting and simplification. The result is manually analyzed to determine whether it yields a new configuration, or (the generalization of) an already existing one. The process terminates when the diagram is closed.

In the *stable* configuration, all faulty processors have been diagnosed. They have remove themselves from their membership sets. The membership sets of non-faulty processors only contain non-faulty processors, and includes all of them. The *latent* configuration corresponds to the arrival of a new fault. It will affect a so far non-faulty processor (x) at the next step (see the outcoming transitions of *latent*). Every path of the type stable $\longrightarrow$ latent $\longrightarrow \ldots \longrightarrow$ stable characterizes a scenario of fault diagnosis.

The complete proof of the GMP involves the following proof steps:

	type id	manifestation of the fault
Send fault	*no_msg*	the other processors q receive : $\neg arrives_q^t \wedge null_q^t$
	not_no_msg	the other processors q receive: $\neg arrives_q^t \wedge \neg null_q^t$
Receive fault	*null*	whatever the broadcaster's behavior, p receives: $\neg arrives_p^t \wedge null_p^t$
	not_null	whatever the broadcaster's behavior, p receives: $\neg arrives_p^t \wedge \neg null_p^t$

Figure 3. Fault model for the test experiments

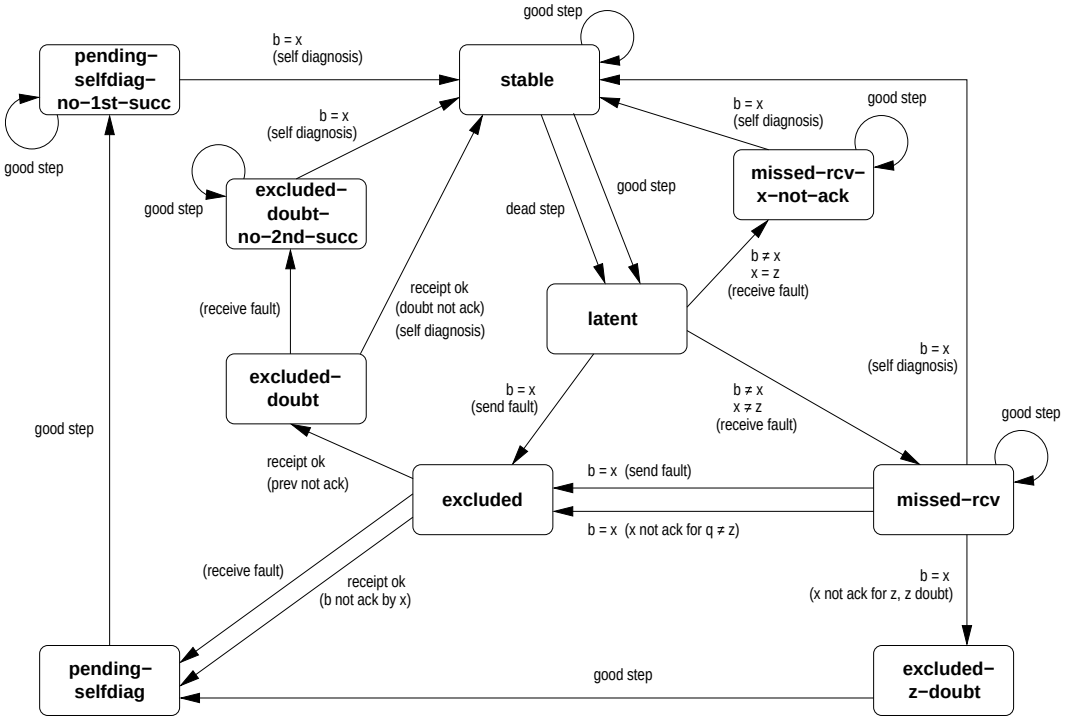

Figure 4. Configuration Diagram from [8]

- prove that the validity and agreement properties hold in every configuration.
- prove the transition lemmas. For each transition, it is proved that starting from the source configuration at time `t`, if the transition condition holds, then the system will be in the sink configuration at time `t+1`.
- prove that there is no other configuration the system can get into. It is first proved that the stable configuration initially holds. Then, for every configuration, it is proved that the specification of transitions is complete.
- prove the self-diagnosis property. This is done by first proving that the system remains outside the stable configuration for at most `2n` slots.

For the GMP version presented in [8], all these proof steps were successfully discharged under the PVS environment. They correspond to 348 proof obligations.

6. Flaw insertion

For experimentation purposes, we need to insert a flaw in the algorithm and obtain a partial proof.

6.1. Choice of the flaw to be inserted

A total amount of 10 candidate modifications of the algorithm were considered. We first considered a few Mutation-like [1] modifications, that consist in simple syntactic changes in the algorithm. Then we identified other candidate modifications by searching for discrepancies between: 1) the PVS source code of the algorithm and its informal presentation in [8]; 2) the current version of the algorithm and previous versions presented in [7, 10]; 3) the studied GMP and a related GMP algorithm, proposed in [2] (attempts to prove the latter algorithm led the disjunctive invariants proof method to be developed, as mentioned in Sections 2 and 5.1).

For our purpose, a candidate modification should be retained only if it possesses some desired characteristics. The modification should not yield a crudely incorrect algorithm. Still, it should correspond to a flaw: we are not interested in modifications preserving correctness with respect to the three properties of validity, agreement and self-diagnosis. Also, obtaining a realistic partial proof should not necessitate major changes in the original proof, so as to keep the effort reasonable.

Determining whether a candidate modification possesses the desired characteristics is obviously a problem. Crudely incorrect algorithms can be identified by test experiments under the blind random profile developed at the end of the high-level analysis (see section 4.3). Such preliminary experiments allowed us to eliminate five candidate modifications yielding a high failure rate. For the other modifications yielding no failure (4 modifications), or few failures (1

modification), further analysis of their characteristics had to involve formal reworking.

It turned out that we obtained a complete proof for two of the modifications for which no failure was observed. One of them corresponds of a mutation affecting command (7) of the GMP: in the action part, the incrementation of the rej counter is suppressed (see the original command in Figure 2). This example illustrates the difficulty of understanding the semantic impact of a modification. The fact that the algorithm still works is far from intuitive.

We did not manage to complete the proof of the two other modifications yielding no failure. We were not able to determine whether the modifications preserve the three required properties, or correspond to subtle flaws. In one case, the partial proof we obtain after some formal reworking is not deemed representative of a genuine attempt to prove the modified algorithm. Our opinion is that a major change in the configuration diagram (and hence in the proof structure) would be required to properly account for the algorithm's modification. As a result, this modification is not retained. For the remaining modification yielding no failure, as well as for the one inducing a low failure rate under the crude profile, we managed to obtain meaningful partial proofs at the expense of a reasonable effort.

For first experimentation, we decided to retain the modification known to introduce a flaw (i.e., the one which failed under the crude profile). The flaw, as well as the obtained partial proof, are presented in the next section.

6.2. Retained modification of the GMP

The inserted flaw induces a low probability of failure under the crude random profile (0.6%). It consists in weakening the guard of Command (1) of the algorithm:

$$acc_p^t > rej_p^t \wedge \quad \text{is turned to} \quad acc_p^t > rej_p^t$$
$$acc_p^t \geq 2$$

The second part of the guard ($acc_p^t \geq 2$) corresponds to one of the modifications introduced between early PVS versions of the algorithm [7, 10] and the most recent one [8]. The author of the proof identified the modification as necessary to avoid failure in case of a specific scenario. This scenario corresponds to a specific activation of path: stable $\rightarrow$ latent $\rightarrow$ missed-rcv-x-not-ack $\rightarrow$ stable. The path is triggered by a receive fault on the most recent broadcater x, and the specific activation is when x fails to detect any communication at all during the next `n-1` slots (according to our terminology, it makes `n-1` successive $null$ receive faults).
In the correct version of the GMP, the last transition of the path is taken as x becomes broadcaster again. It executes command (2), because the guard of (1) evaluates to false. In the flawed version, command (1) is executed, processor x does not diagnose its fault, and the system behavior goes outside the configuration diagram. The self-diagnosis property is violated.

In order to obtain a realistic partial proof of the flawed algorithm, it is not sufficient to modify the PVS description of the algorithm. The modification has to be propagated. It has to be accounted for in other parts of the PVS specification, as well as in their proofs. A first work on definitions and lemmas allowed us to reconstruct the original proof, with the exception of three pending lemmas. The three lemmas correspond to the proofs of the three following transitions:

- missed-rcv-x-not-ack $\rightarrow$ stable
- excluded-doubt-no-2nd-succ $\rightarrow$ stable
- pending-selfdiag-no-1st-succ $\rightarrow$ stable

After analysis, we concluded that the second transition could be proved at the expense of a minor modification of the configuration diagram. Indeed, after having strengthened the predicates defining configurations excluded, excluded-doubt, excluded-doubt-no-2nd-succ, and after having reworked all transitions proofs linked to these configurations, we managed to complete the proof of excluded-doubt-no-2nd-succ $\rightarrow$ stable.

At this stage, there are two pending transitions in the GMP proof for which no obvious solution can be found. This not surprising for the missed-rcv-x-not-ack $\rightarrow$ stable transition, as the known revealing scenario is related to it. But the proof is also unsuccessful in the case of the other transition, for which we do not have any counter-example.

In our opinion, this partial proof can be seen as a realistic attempt to prove the modified algorithm. Thus, it will be used as a basis to guide the design of testing.

7. Proof-guided testing

After a brief discussion on the principle of proof-guided testing (Section 7.1), we give experimental results corresponding to the two levels of analysis of the proof: analysis at the lemma level (Section 7.2), and analysis at the sequent level (Section 7.3).

7.1. Principle

Our approach relies on the assumption that the identification of the pending parts of the proof should supply useful information for the design of testing. More precisely, the aim is to trigger a violation of the required GMP properties, and we will try to achieve this by means of a falsification of the pending parts of the proof.

Of course, falsifying the pending parts of a proof is not necessarily a practical objective for testing. Undecidability problems, as well as the introduction of auxiliary formulas as proof artifacts, may result in pending parts that are neither controllable nor observable. In the worst case, when no

constructive information can be extracted from the proof, we are in the same situation as at the end of the high level analysis, that is:

- Violation of any one of the required properties is observable (by means of the implemented oracle checks).
- Violation of the properties is not specifically controllable. However, we are able to exhibit an input generation function such that, should a violation be possible, revealing inputs would have a non null probability of being generated (implementation of the crude random profile).

In practice, the design of testing is improved by identifying subdomains that can safely be removed from the test input domain, and by trying to define a meaningful distribution of probabilities over the remaining domain, based on the proof structure. This calls for understanding the proof structure, and for being able to establish a link between proof parts and the operational behavior of the algorithm.

The analysis can be conducted at different levels. One may simply consider the fact that two transition lemmas are pending. One may also refine the analysis and consider the details of the proof trees attempting to discharge each transition lemma.

Conducting analysis at the lemma level does not require high expertise[3]. It is sufficient to be able to read the PVS specification language, so as to understand the global proof structure (i.e. understand the definition of the configuration diagram). The extraction of constructive information is then facilitated by the fact that the configuration diagrams provides an operational view of the algorithm's behavior: testing can be directed toward the activation of the two pending transition lemmas.

Conducting analysis at the sequent level is more demanding. It requires some expertise in the PVS prover, in order to understand the proof trees and analyze their pending sequents. Establishing a link between the sequents and pieces of operational behavior is also expected to be much more difficult than at the lemma level.

Both levels of analysis were considered for deriving test sets.

7.2. Test criterion at the lemma level

At the lemma level, the retained test criterion is the coverage of all paths stable $\rightarrow \ldots \rightarrow$ stable that may trigger the activation of unproved transitions. Note that there are 20 feasible paths in the complete diagram; 14 of them include one the target transitions.

[3]In our case, the flaw insertion process necessitated the rework of the proof. But in the case of a genuine partial proof, the tester would simply use the raw results of the proof in terms of pending lemmas

We designed a sampling profile that makes the relevant paths roughly equally likely. Actually, the profile is only an approximation of an equiprobable one. This is so because, in the PVS specification, the transition conditions cannot always be easily linked to input cases: they also depend on internal variables of the model. Hence, we only have an imperfect control of path coverage. Note that a few generated sequences may fail to activate the target transitions; however, the definition of the profile ensures that no test sequence covering the paths of interest has a null probability of selection.

The adequacy of the retained criterion was assessed by testing the flawed algorithm with a large (50,000) sample of sequences generated under this profile. We obtained the following results:

- 0.9% of the generated sequences yielded a failure of the algorithm.
- Like in the crude random profile, all failures corresponded to a violation of the self-diagnosis property.

Let us recall that under the crude random profile, the failure rate is 0.6%.

Two conclusions can be drawn from the analysis of the results:

- Strictly speaking, the information extracted from the proof is not irrelevant for revealing the flaw. Whatever the profile (including the crude one), all observed failures correspond to sequences that do activate the target transitions.
- While not irrelevant, the information is still quite imperfect. In particular, deterministic selection of one test sequence for each of the 14 paths of interest would yield a low probability of revealing the flaw. As regards statistical testing, the designed profile only supplies a modest improvement with respect to the blind one. A sample of 345 sequences is required to get a 0.95 probability of revealing the flaw.

We now consider the detailed analysis of the proof trees to refine the test criterion.

7.3. Test criterion at the proof tree level

Both transition proofs face a similar problem. In the proof trees, there are undischarged sequents corresponding to the proof of goal $acc_x^t \leq rej_x^t$ under hypothesis $acc_x^t = 1$. The definitions of configurations missed-rcv-x-not-ack and pending-selfdiag-no-1st-succ ensure that $acc_x^t = 1$ holds. Hence, we should try to falsify the pending sequents with $rej_x^t = 0$.

This objective has to put in the form of operational test sequences. We managed to do this by referring to the algorithm's specification. We know that the rej counter was

reset to zero by command (1) at the last broadcast of x. It is thus required that, since its last broadcast, x has never executed a command incrementing its rej counter. Looking at the algorithm, we can conclude that:

- x has never executed commands (5), (7), (11), (14) since its last broadcast.
- Since $acc = 1$ in the target configurations, it also has never executed commands (4), (8), (9), (12).
- It also has never executed commands (3), because in the target configurations x is specified to be included in its own membership set.

Hence, the only commands x has executed are commands (6), (10) or (13), which means that x has never detected activity on the bus since its last broadcast (the $null$ input is true at each step).

The selection criteria derived from the lemma analysis is then refined, by removing the paths that do not conform to this requirement, and by restricting the input subspaces of the remaining paths: they will be covered by test sequences with suffix including only $null$ receive faults (the size of the suffix depends on the selected path).

Under this profile, a sample of 50,000 random sequences was generated. It supplies the following results:

- The failure rate is now 98.7%.
- As previously, all failures correspond to a violation of the self-diagnosis property.
- The non-revealing sequences correspond to the few sequences failing to activate the paths of interest, and result from the imperfect control we have on path coverage (like in the previous profile).

The detailed analysis at the sequent level allowed us to focus testing on revealing subdomains. We have a perfect connection of the selection criterion with the flaw residing in the algorithm. Note that revealing sequences for transition missed-rcv-x-not-ack $\rightarrow$ stable are similar to the scenario already identified by H. Pfeifer (see Section 6.2). To the best of our knowledge, fault scenarios for transition pending-selfdiag-no-1st-succ $\rightarrow$ stable are new. These scenarios are longer than the previous one. We claim that they would have been difficult to invent by hand analysis. In [8], the author of the proof mentioned that the GMP actually removes faulty processors more quickly than the proved bound ($2n - 1$ steps). He conjectured that the actual bound should roughly be one and a half round (a round is n steps). But for some of the revealing test sequences we generated, self-diagnosis requires $2n - 2$ steps on the correct version of the algorithm.

8. Conclusion

The results show that proof-guided testing can be very effective for revealing flaws in the case of a partial proof. It can be a pragmatic approach in order to exhibit counter-examples in cases where model-checking would be difficult to apply (as for the GMP example, see the proof vs. model-checking discussion in [11]). In this way, the effort that was put into the proof development is not lost, and testing is directed to the revealing of flaws that were not caught by the partial proof.

However, a deep analysis of the proof may be required. From our experience, detailed analysis at the sequent level represents a significant effort. We recommend that selection criteria based on lemma analysis be first tried. The sequent analysis should be performed only if large samples of random sequences fail to reveal a flaw. In this case, in order to be able to refine the test criterion, it is necessary either to have detailed documentation of the proof (as we had in [8]) or to work in close collaboration with the persons having developed the proof.

We are aware that our results need to be consolidated by further experimentation. We are currently studying other examples of modifications of the GMP algorithm (yielding flaws). But more importantly, there will be a need for experimenting with other examples of proof approaches for realistic FT algorithms.

The proof approach used for the GMP, based on a diagram configuration, turned out to be very adequate from the perspective of testing. Since the proof structure is based on an operational view of the algorithm's behavior, it was possible to establish a link between pending parts of the proof and functional cases for the algorithm. For other proof approaches, e.g. more traditional proofs through invariant strengthening, establishing such a link might be more difficult.

Acknowledgement

We would like to thank Holger Pfeifer very much. He kindly accepted to send us his PVS source files, as well as his Thesis ([8]) chapter describing his formal specification and proof of the GMP.

References

[1] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[2] S. Katz, P. Lincoln, and J. Rushby. Low-overhead time-triggered group membership. In M. Mavronicolas and P. Tsigas, editors, *11^{th} Int. Workshop on Distributed Algorithms*

(WDAG'97), pages 155–169, Saarbrücken Germany, Sept. 1997. Springer-Verlag. LNCS 1320.

[3] H. Kopetz and G. Grünsteidl. TTP – a time-triggered protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, Jan. 1994.

[4] G. Lussier and H. Waeselynck. Informal Proof Analysis Towards Testing Enhancement. In *13^{th} Int. Symposium on Software Reliability Engineering (ISSRE'02)*, pages 27–38, Annapolis, MD, USA, Nov. 2002. IEEE Computer Society.

[5] G. Lussier, H. Waeselynck, and K. Guennoun. Proof-guided testing: an experimental study. In *28^{th} International Computer Software and Applications Conference (COMPSAC'04)*. LAAS research report n° 04154, 2004. *To appear*.

[6] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Engineering*, 21(2):107–125, Feb. 1995.

[7] H. Pfeifer. Formal verification of the TTP group membership algorithm. In T. Bolognesi and D. Latella, editors, *Formal Methods for Distributed System Development Proceedings of FORTE XIII / PSTV XX 2000*, pages 3–18, Pisa, Italy, Oct. 2000. Kluwer Academic Publishers.

[8] H. Pfeifer. *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture*. PhD thesis, Universität Ulm, Germany, 2003.

[9] H. Pfeifer, D. Schwier, and F. W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In C. B. Weinstock and J. R. (eds.), editors, *7^{th} Dependable Computing for Critical Applications (DCCA'99)*, volume 12 of *Dependable Computing and Fault-Tolerant Systems*, pages 207–226. IEEE Computer Society, jan 1999.

[10] H. Pfeifer and F. W. von Henke. Formal Analysis for Dependability Properties: the Time-Triggered Architecture Example. In *8^{th} IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'01)*, pages 343–352, Antibes Juan-les-Pins, Oct. 2001. IEEE.

[11] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification (CAV'00)*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag. LNCS 1855.

[12] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. Software staistical testing. In H. B. Randell, J-C. Laprie and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 253–272. Springer Verlag, 1995.