Performance Validation of Fault-Tolerance Software: a Compositional Approach

Simona Bernardi, Susanna Donatelli
Dipartimento di Informatica, Università di Torino, Italy
Corso Svizzera 185, 10149 Torino, Italy
{bernardi, susi}@di.unito.it

Abstract

This paper discusses the lessons learned in the modeling of a software fault tolerance solution built by a consortium of universities and industrial companies for an Esprit project called TIRAN. The requirements of high flexibility and modularity for the software have lead to a modeling approach strongly based on compositionality. Since the interest was for assessing both correctness and performance of the proposed solution, we have cared for these two aspects at the same time, and, by means of an example, we show how this was a central aspect of our analysis.

1. Introduction

In this paper we describe the modeling approach for the study of software fault tolerance strategies used in the TIRAN project [8, 9, 3]. TIRAN (Tailorable fault tolerance framework for embedded applications) is an European ESPRIT project, ended in October 2000, that involved six European partners from industries and universities, that has defined and implemented a software based solution for fault tolerance in embedded systems.

For terminology on fault-tolerance, we adhere to the one proposed by Laprie in [19, 21]. A system failure occurs when the delivered service deviates from fulfilling the system intended function and the failure process of a system is represented by the following fundamental “FEF chain”: fault → error → failure that emphasizes the cause-effect relation among the events that bring the system to a failure. Fault tolerance methods try to ensure that a service will fulfill the system’s function in spite of the presence of faults by carrying out error processing and fault treatment that break the FEF chain [20]. The TIRAN proposed solution to fault tolerance is to provide a framework that consists of a library of functions to add fault tolerant behavior to software, a support for their execution called “backbone”, and a language to specify how to react to errors called “Ariel”. In TIRAN the set of functions that implement a given fault treatment aspect are grouped together under the name of “mechanism”. Example of mechanisms are: the stable memory (to realize temporal and physical redundancy of memory objects without specific hardware), the distributed synchronization (to allow synchronization, over a set of synchronization points, among a set of tasks that can fail and recover), the watchdog (a sophisticated version of a time-out), and a voter (to execute multiple copies of an application and to vote on the application outcomes).

Modeling was used in the project to support the evaluation of the user satisfaction of the TIRAN fault tolerance framework for a given application that uses the framework. Users may accept a given mechanism only if, for a given fault and error hypothesis there is no failure produced (so if the mechanism actually does what it is supposed to) and if it does not impose unacceptable delays (under normal behavior) or unacceptable degradation (under faulty behavior).

In this context the correctness requirements mix up with the performance requirements, moreover it is very important that the verification and evaluation processes take place already at the specification/design phase, since there is obviously no point in implementing solutions that, may be already by design, do not satisfy the user requirements. When evaluation is required before the final product is available, the only feasible technique is modeling, since measurements are feasible only if, at least, a prototype is available.

In the TIRAN context flexibility was a major issue for two reasons: first one is that different partners had to agree on a common definition of the mechanisms, and this has required a number of iterations in the specification and design phase, second one is that the framework has been implemented, and is possibly going to be implemented in the future, on different platforms, that may support different basic services, like, for example, communication. Another important aspect is indeed modularity, since the library of the TIRAN framework comes with a library of mechanisms.
models, to possibly support future performance evaluation activities in the companies that will use the framework to increase the fault tolerance of their systems.

Modeling in TIRAN takes a stochastic approach to the computation of performance indices, and the modeling language used is that of Petri nets in which transition delays are exponentially distributed random variables; in particular we use a class of nets known as GSPN [2]. It is important to remark that stochastic analysis is not adequate to answer questions concerning non-stochastic events, e.g. maximal execution time of a procedure assuming that all delays involved are deterministic. Moreover the analysis does not deal with limited resources (since there is no model of the physical resources) which implies that all results obtained represent the best performance that the software can provide. Although this is not very realistic in absolute terms, it is the only option in a context like TIRAN were emphasis is on software-based fault tolerance, and no hypothesis are made on mapping of processes and mechanisms on nodes. If these informations are available the hardware models could be included using, for example, the compositional approaches in [11].

The GSPN models of the mechanisms have been produced starting from semi-formal specification provided by the TIRAN specification documents (that includes state diagram, message diagram and textual comments). The translation process is not automatic due to a lack of complete formal specifications, but we have adhered to the basic ideas behind the automatic translation of CSP-like programs to nets presented in [4] and of UML state chart, message diagram and collaboration diagram to nets presented in [18].

We have heavily used composition, which is a standard argument in process algebra, and it has gained wide popularity in nets as well [6]. Compositionality for fault tolerance analysis was used also in [16] and [17] where the emphasis is on modeling the dependences between hardware and software components and on studying the relationships between service degradation levels and the architectural failure modes. The approach in [16] and [17] is based on a rich set of compositional rules and interface nets, while we have a single rule, but it seems feasible to compose the two approaches to include both an explicit model of the hardware failures as in [17] and a detailed model of the service model as we do.

Evaluation of software from the specification stage, possibly with automatic translation from specifications to models, has also gained increased popularity in the last years. In all cases that we know, although, the studies are directed either towards correctness analysis, as in e.g. [22], or towards performance evaluation, as in e.g. [18]. In [22] the vUML tool is used to verify state charts of UML specifications: the tool uses the model checker SPIN [15] to verify properties, and counter examples for properties are shown back to the user in UML notation (as sequence diagrams). In [18] the translation is from UML diagram (mainly state charts and collaboration diagrams) to GSPN, and the translation is done clearly in a performance evaluation perspective, although nothing forbids to check properties on the resulting GSPN.

The paper is organized as follows: Section 2 introduces the basic definition of GSPN and of net composition, Section 3 presents the basic idea for the construction of model components and principles of the analysis. Section 4 discusses the application of the general idea to the construction of the watchdog mechanism models, our running example. Section 5 illustrates the analysis of the watchdog models, showing the interplay between correctness results and performance evaluation, while Section 6 summarizes the paper and discusses open points of the proposed approach.

2. Definitions and notation

Generalized SPN (GSPN) were originally proposed in [1] to support the performance evaluation of concurrent and distributed systems. We assume the reader is familiar with the formalism, and we introduce them here only to fix the notation. A complete introduction can instead be found in [2].

GSPN models comprise two types of transitions: timed transitions, which are associated with random, exponentially distributed firing delays, and immediate transitions, which fire in zero time, with priority over timed transitions. A GSPN is thus an eight-tuple $GSPN = (P, T, I, O, H, M_0, W)$ where $P$ is the set of places, $T$ is the set of transitions, $I, O, H : T \rightarrow \mathbb{N}$, are the input, output and inhibitor functions, $M_0$ is the initial marking, $\Pi : T \rightarrow \mathbb{N}$ is the priority function which associates lowest priority (0) with timed transitions and higher priorities (≥ 1) with immediate transitions, $W : T \rightarrow \mathbb{R}$ is the weight function: $W(t)$ is the parameter of the negative exponential probability density function of the transition firing delay, if $t$ is a timed transition, or the weight used for the computation of firing probabilities of immediate transitions, if $t$ is an immediate transition. The set of reachable states can be partitioned according to the type of transitions that are enabled in a state: if the state enables no immediate transitions the state is termed tangible, if instead an immediate is enabled the state is vanishing. The reachability graph can then be reduced to the set of tangible states only, to produce a tangible reachability graph (TRG), that is isomorphic to the Continuous Time Markov Chain (CTMC) that describes the stochastic process of the GSPN.

In the graphical representation used in this paper, immediate transitions are drawn as segments, and exponential transitions as white rectangular boxes.

A labeled GSPN system (LGSPN) is defined as $LGSPN = \{ P, T, I, O, H, M_0, W, L \}$ where $L : P \rightarrow \mathbb{N}$ is the notation. A complete introduction can instead be found in [2].
(λ, G) where G is a GSPN system, and λ : T → L ∪ τ, assigns a label from a set L, or a τ label to each transition. τ labeled transitions are considered to be internal, non-τ are considered external (i.e., communication transitions).

In the graphical representation labels are associated to transition names, separated by a vertical bar, τ labels are usually not written.

![Figure 1. Transition superposition.](image)

To compose labeled GSPN we shall use the classical [6, 7] superposition of transitions of equal label. Given two GSPN systems LG1 and LG2 we denote the LGSPN system LG, transition superposition of LG1 and LG2 over the set LM ⊆ L of labels, as LG = LG1 ||L LG2. LM is called the “synchronization set”.

An example of transition superposition is illustrated in Figure 1, where LG1 is composed with LG2 over the set L = {a}. The resulting net contains, for each label in the synchronization set, the cross-product of the transitions of equal label. It is obvious that the ||L operator realizes a “rendez-vous like” synchronization among the two nets.

Observe that the labeling function does not need to be injective, and that transitions that represent the superposition over a label l are labeled l (and not τ) in the composed net. A formal definition of the ||L operator can be found in [11]. The operator can be extended in a straightforward manner to the case of more than one label associated to a transition (multi-labeling), that is to say with a labeling function λ : T → P(L) ∪ τ, where P(L) is the power-set of L.

Performance evaluation of GSPN and labeled GSPN models can take advantage of the steady state and transient solution of the associated CTMC, as well as of discrete event simulation, if the state space is too large or non-exponential distributions are freely used. Verification of correctness properties can be based on structural properties of the net, like P- and T-invariants, or on state space exploration, possibly with sophisticated methods as the one used for model checking [23].

3. Model construction and analysis

The TIRAN framework is organized into three layers: a basic layer, with mechanisms related to error detection, fault containment, fault masking and recovery, a control layer (backbone) for a coherent coordination of the entities of the basic layer and, possibly, of the applications, and a monitoring and fault-injection layer, for testing purposes. In this paper we consider only the two first layers, which are the one that influence the application.

3.1. Building blocks

Models are built out of basic component models: mechanism, user, communication, fault, and backbone. The first component is the mechanism model, that can be derived from the specification documents. To “exercise” the mechanism model we need another model that interacts with it, to represents a “typical behavior” of an application that uses the mechanism: this model will be called UM model in the following. Of course there is one UM model per mechanism.

The Fault Generator model (FG), depicted in the non-shadowed portion of Figure 2, is an elaboration of previous PN models of faults proposed in [24] (that are, in turn, rather similar to the one proposed in [12] for coverage modeling) where faults are classified with respect to their persistence in permanent, intermittent and transient.

![Figure 2. The fault model.](image)

Permanent fault is charged with obsolete or worn out internal physical components and it occurs when those components are activated by the application process. Once it has occurred it remains always active: an active permanent fault is modeled by place act_pf marked. Intermittent fault becomes active depending whether a particular system condition holds. Once it has occurred, it is characterized by alternating periods in which it is active (place act_if is marked) and it may lead the application process to an error state, and periods in which it is latent (place lat_if is marked) and hence it does not cause any error in the application process. Transient fault is a fault that, once it has occurred, it remains active for a certain amount of time and then it disappears. Observe that, as in [24], permanent and intermittent are modeled as mutually exclusive events.

FG interacts with the user application model UM through transitions labeled flat whose firing delays represent the fault latency, i.e. the elapsing of time between the oc-
currence of a fault and the appearance of the corresponding error in the application process. We assume a single fault occurrence during the execution of the application process, i.e. we do not consider faults related or connected to the first one that make worse the state of the system.

The backbone model BK represents the behavior of the control layer. This behavior is actually driven by the recovery language specification, so in the model we comprise a set of “default actions” associated with each notification from the mechanisms. Indeed the TIRAN framework specification document did not contain any information about default actions from the backbone, although each specification engineering had in mind a typical behavior for BK (for example some of the mechanisms implicitly assume that a notification to the backbone must be followed by an acknowledgment message) and it was actually a merit of the modeling activity work to show that correctness could be proved only under a precise hypothesis on the backbone behavior.

The component models interact through a model of the communication; to experiment with different types of communication we consider communication as a “service” for which there is a given implementation, possibly changing from platform to platform. By composing the user model and the application model with different communication nets, we can exercise our model under different hypothesis. To illustrate the concept we consider here four cases: rendez-vous communication with transmission in zero or non zero delay, and asynchronous (mailbox based) send with blocking receive and possibly a time-out.

Figure 3 shows a rendez-vous communication over a channel A: component P1 wants to send a message to P2 over A and P2 wants to receive it, the two models can then be composed with a model S1, that is a single transition labeled with both the send label and the receive label (multi-labeling). The definition of the overall system Sys takes the parallel composition of P1 and P2 (the superpose operator applied over an empty set of common labels), and composes it with the communication service S1 over the set of labels \{sendA, recvA\}. As a consequence of the composition, the three transitions are superposed into a single one, thus correctly modeling a strict rendez-vous. Observe that, if process P1 has two transitions labeled sendA and P2 has three transitions labeled recvA, then the synchronization of P1, P2, and S1 produces six transitions, representing all possible combinations of send and receive. A similar result can be obtained by labeling the transitions of P1, P2, and S1 with the same label.

The case of extended rendez-vous is instead presented in Figure 4, starting from the same two processes P1 and P2, each transition labeled with a label from the superposition set is expanded into pairs representing the start and the end of the communication. The resulting models are then superposed with the communication service S1, that explicitly represents the communication time. The combination of the expansion with the transition superposition was named “client-server cooperation” in [11].

Figure 4. Synchronous case - with time

The case of asynchronous (mailbox based) communication is shown in Figure 5. The communication in P1 and P2 is modeled as before, while the communication service has changed to account for the presence of the mailbox. Service S1 implements an asynchronous send with blocking receive, while service S2 implements an asynchronous send with non-blocking receive, thanks to the presence of an additional transition labeled recvA in S2, that is enabled only when the mailbox is empty: in this case the labels in P1 and
When no fault takes place, and overhead on MET when a tolerated fault takes place.

Another important goal is to establish the type of faults/error that an application can tolerate, and this is done studying the functional behavior of the model. In the project we have often used deadlock analysis of open models that describe a single run of the application and/or a single run of the mechanism.

A number of additional goals are instead “mechanism-specific”: for error detection mechanisms like the watchdog analyzed in this paper we can compute the percentage of false error detection and the mean time to detect an error (how long does it take for the mechanism to detect the error once it has happened, and to signal the backbone). Measures for masking mechanisms like the n-out-of-m voter [25] are the percentage of correct computations and mean time to failure (MTTF) for cyclic applications, while for synchronization mechanisms it could be of interest to investigate correctness in terms of absence of deadlock even in the presence of tasks that may be faulty and restarted.

4. The models of the watchdog example

Figures 6, 7, and 8 depict the models of the watchdog mechanism, of its user model and of the backbone respectively, that, together with the FG model and one of the communication models of the previous section, allow to study the watchdog mechanism (WD from now on). For this section, only the non-shadowed portions of the nets shown in the figures are relevant. Moreover we have used the convention that labels Sxx and Rxx represent matching pairs of send and receive (either through mailbox xx or in a rendez-vous fashion, depending on the communication model). Since in most cases we do not make hypothesis on the type of communication, we shall refer to xx as either “message xx” or “action xx”.

The watchdog mechanism of TIRAN is a time-out server that offers to an application the possibility of setting a time-out, of pausing it, and resetting it (kick). If the time-out expires a message is sent to the backbone.

The WD model, shown in Figure 6, has been derived from the automata representation of the specification document. WD is activated by a setup message, and, after performing a setup operation, goes into a pause state, from which, upon reception of a continue message (with the associated continue operation), and after having sent back an ack, it goes into a counting state (place count marked). In this state the physical timer is set to its initial value and starts the count-down (represented by the timed transition alarm). While the timer is counting-down, WD may receive a kick message, that cause WD to reset its timer and to go back to place count, or a pause, that causes WD to stop the timer and to go back to place pause. When the timed
transition *alarm* fires the WD sends a message to the backbone and then it moves to the expired notified state (place *exp_notified*) waiting for an ack message from the backbone. After the reception of the ack it will move to its initial state (place *unused*). WD can terminate (upon reception of a *terminate* message) only when it is in state *count*.

The application model (or user model UM) initializes the WD with a *setup*, followed by a pair of message exchanges with WD (*continue* and *ackcontinue*) to model the request for a count-down to start. The timed transition *activity* models the computation that the application wants to protect by WD. When the activity is terminated the application can terminate the WD to then cycle back to the initial state *start*, or it can perform a new computation. This computation can be again protected by WD (transition *again*), or it is an external operation of unknown delay, for which the WD is not used (transition *extop*). For protecting there is the choice between restarting the counter (*kick*), or using the remaining value (transition *nokick*). For the unprotected choice, the application needs to pause WD (action *pause*), and then the computation can be performed (transition *extop*).

According to the watchdog specification certain messages require an acknowledgment to be sent back, from the watchdog to the application. This is indeed the case of *continue*, for which there is an explicit action of acknowledge (*ackcontinue*), both in UM and WD.

The GSPN model of the backbone behavior is shown in Figure 8, (non-shadowed portion): when a *notifyBB* message arrives, an *acknotifyBB* is sent back.

5. Analysis

The analysis is split in two phases: the first one considers the “normal behavior” of the system, assuming that there are no faults, while the second one considers a system that can experience a fault.

5.1. Overhead when no faults are present

The goal of the analysis is to study the overhead of the application when the watchdog is used but no faults (and therefore errors) are ever generated. The complete model is obtained composing the model of the application shown in Figure 7, non-shadowed portion, the watchdog of Figure 6, non-shadowed portion, and a rendez-vous model for the communication over the matching R and S labels *setup*, *continue*, *ackcontinue*, *terminate*, *kick*, and *pause*, that is to say for all communications between the two models. No model of fault is included, and no model of backbone (since there are no faults the backbone should never enter into play). Since no backbone model is included, labels *notifyBB* and *acknotifyBB* become r. We shall call this model Mdl-1.

A correctness criteria for the watchdog is that, when no fault is present, the time-out never fires, and that the application can go into an infinite series of *activity*. Neither
of these conditions is true as it was proved by the correctness analysis that uses Mdl-1 modified so as to make it open (to model a single run of UM) by changing the arc from t4 back to place start into a new place end. The generation of the state space results in 16 tangible states, with four deadlocks, one represents a normal termination of the application (places end of UM and unused of WD both marked), while in the remaining ones place expired of WD is marked, meaning that the alarm transition has fired, although no fault was present (false alarm).

Observe that, if we use instead a mailbox communication model for the composition, and if we take a mailbox per each different label, the mailbox used for exchanging the kick message is not bounded, and this indicates an error in the specifications, that are changed to require that a kick be followed by an acknowledge message. If we limit the mailbox place artificially to one token, we detect 7 deadlocks, that basically have the same interpretation as the four above.

To compute the overhead in absence of faults it is therefore necessary to ensure that, even if the timeout expires due to a false alarm, the model goes back into an operational state. This is achieved by having the backbone model coming into play upon notification from the watchdog, to take the watchdog back to the count place through the firing of transition t7 of Figure 6 modified to go into place count. A new model Mdl-2 is produced: it has 15 tangible states and transition alarm has a non-zero throughput, so that false alarms are generated, but no deadlock is present. Transition activity is reported to be live, and therefore can be executed infinitely often.

The mean execution time of the application (MET) is defined by $\frac{1}{\text{th}(\text{end})}$, where $\text{th}()$ stands for “throughput”. Observe that this represents a correct application of Little’s formula, since there is a P-invariant that states that the sum of tokens over all places of UM is equal to 1.

To compute the overhead due to the presence of the watchdog mechanism we need to compute the MET index also for the application without the use of the mechanism. For this computation we have used the model of Figure 7 alone, without any composition with the WD model and the communication model.

The small dimension of Mdl-2 allows to easily perform sensitivity analysis for measures like, for example, the frequency of a false alarm for varying values of the rate associated to transition alarm. Moreover it is easy to compare the performance under different communication hypothesis. Figures 9 and 10 show, respectively, the MET of the application and the frequency of false alarms, expressed in terms of throughput of transition alarm, in case of synchronous and asynchronous communications. Additional results and different rates assignments can be found in the TIRAN technical report [5].

5.2. Analysis in the presence of faults

For level II analysis we have to add the model of faults FG. A fault becomes an error when it changes the state of the application (when it is perceived by the application) and an error from which it is not possible to recover is considered a failure. To build a model that considers faults, a few important aspects have to be considered:

1. how do the components perceive a fault?
2. does a fault always provoke an error?
3. which are the components that can go into error?
This means that given UM goes into error state, WD goes into expired, and therefore the error is detected.

For what concerns performance criteria, an interesting measure for the application is the distribution of “Time To First Detected Error” (TTFDE) and “Time To First Undetected Error (TTFUE)”, and corresponding mean values, computed using transient analysis since the model is open. For the first measure we can define the state in which places errorstate and unused are marked as a “failure state” and we can compute a standard “time to failure” index. Places errorstate and pause are instead used for TTFUE.

As a final measure we need to compute the MET overhead in the presence of fault that are correctly treated by the framework, that, in our case, translate into saying that the watchdog is never paused. In this case steady state analysis can be performed on an ergodic model, using the complete UM model of Figure 7 without pause branch, the shadowed backbone model of Figure 8, the complete fault model of Figure 2 and the complete watchdog model of Figure 6. Results for this model can be found in [5].

6 Conclusions

Modularity, flexibility, integration between verification results and performance evaluation, are the main keywords of this work. In this paper we have shown how net composition can support flexibility, which is a must when the evaluation is performed at the specification phase or at the early-design stage. We have placed our work in the context of an Esprit project in the software fault-tolerance field, in which a number of software solutions had to be evaluated in terms of correctness and performance. We think our case study provides an interesting example of “global evaluation” in which correctness aspects (that advocates verification) and performance aspects (that advocates performance evaluation) goes hands in hands in a synergic manner.

Models were produced from the semi-formal specification of the mechanisms and of the usage of the mechanisms by the end-user. Since the only formal part of the specification where state diagrams, and message charts (UML was taken as a reference point by the consortium partners, but not as a must), our models are rather detailed. This was not
a problem for the watchdog mechanism, but it is actually a problem for more complicated ones, especially for what concerns the performance evaluation part. Apart from the obvious possibility of using simulation, we plan to work on the possibility of an automatic reduction of models, by declaring certain actions as being of negligible relevance with respect to time, transforming them into immediate, \( \tau \)-labeled, to then apply classical equivalence techniques based on observability. For the watchdog case we can ask ourselves whether it is possible, from the models described in this paper, to derive the “classical” performance evaluation model of a time-out, in which a timed transition representing the application is conflicting with the alarm transition. Another possible approach is to produce directly an abstract model that describes the mechanism in a very synthetic manner, and is derived from a natural language high level description of the mechanism, instead than from the specification documents. Abstract models can be used for a first and cheap evaluation of the mechanism, to guide the search space of the detailed models, although they are of no help for assessing correctness.

A problem that we have not been addressed in this paper is that of the assignment of rates to transitions, which, in a modular context, gets even more complicated, as the work on stochastic process algebra has shown [14]. In TIRAN models are first composed and then rates are assigned, according to range indications coming from the industrial partners. The same is true for model properties and performance indices, that are always defined on the “final” composed model.

Another related problem is that of assigning a distribution for delays, since it is obvious that, in many cases, exponential is not an appropriate choice. The input from the specification teams indicated an interest for low-variance solutions. Deadlock analysis was also done in GreatSPN, while properties on the state space were checked using PROD [26]: an automatic translator from GreatSPN nets to PROD nets was developed for the project, and was actually initially tested using the nets presented in this paper. Compositionality is not primitive in GreatSPN so that a program called algebra [25] has been built that implements transition and also place superposition using and producing GreatSPN format. Moreover a Java graphical interface has been added to GreatSPN to run multiple experiments and draw curves in an easy manner. Although the example presented in this paper uses GSPN, many of the modeling activities in TIRAN have used high level Petri nets, to limit the graphical complexity, and to take advantage of their efficient solution based on symmetries. Indeed all tools developed for TIRAN (algebra, the PROD translator, and the multisolve) also work in the colored world of SWN.

References


