Integrating Fault-Tolerant Techniques into the Design of Critical Systems[*]

Ricardo J. Rodríguez and José Merseguer

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Zaragoza, Spain
{rjrodriguez, jmerse}@unizar.es

Abstract. Software designs equipped with specification of dependability techniques can help engineers to develop critical systems. In this work, we start to envision how a software engineer can assess that a given dependability technique is adequate for a given software design, i.e., if the technique, when applied, will cause the system to meet a dependability requirement (e.g., an availability degree). So, the idea here presented is how to integrate already developed fault-tolerant techniques in software designs for their analysis. On the one hand, we will assume software behavioural designs as a set of UML state-charts properly annotated with profiles to take into account its performance, dependability and security characteristics, i.e., those properties that may hamper a critical system. On the other hand, we will propose UML models for well-known fault-tolerant techniques. Then, the challenge is how to combine both (the software design and the FT techniques) to assist the software engineer. We will propose to accomplish it through a formal model, in terms of Petri nets, that offers results early in the life-cycle.

1 Introduction

Software failures chronically occur and in most cases do not cause damage. However, a system is called *critical* when failures result in environmental damage (*safety-critical*), in a non-achieved goal compromising the system (*mission-critical*) or in financial losses (*business-critical*). Avizienis et al. [1] cleverly identified the fault-error-failure chain to support specification of intricacies occurring in critical systems.

Fault prevention and fault tolerance, as two of the means to attain dependability [1], have to be considered by designers of critical systems. The former, for example, by means of quality control techniques, while the latter may take the form of replication: distribution through replication confers tolerance to the system and allows to get a higher system availability.

This paper addresses the issue of integrating already developed fault-tolerant (FT) techniques into software designs for their analysis through automatically

[*] This work has been supported by the European Community's Seventh Framework Programme under project DISC (Grant Agreement n.INFSO-ICT-224498) and by the project DPI2006-15390 of the Spanish Ministry of Science and Technology.

obtained formal models. The aim is to evaluate the effectiveness of a given FT technique for a concrete software design, i.e., to verify if the design meets dependability requirements using such FT technique. FT techniques have to be efficaciously integrated with other system requirements, and this will be accomplished through the software design. In fact, we propose to represent a given FT technique as a UML (Unified Modelling Language [2]) model with well-defined interfaces. So, the software design of the critical system under analysis, also modelled with UML, will be equipped with the FT while appropriate interfaces are provided. The software design and the FT technique are eventually converted into blocks of Petri nets [3] (using well-known translation approaches [4,5]) and composed to get the desired analysable model that will report results about system dependability.

The analysis of dependability requirements compels to capture dependability properties (e.g., fault or failure description), which should be expressed in UML designs assuming that we desire to free the software engineer from the manual generation of the formal model. On the one hand, in this work we use DAM [6] (Dependability Analysis and Modelling profile) for this purpose. DAM is a MARTE [7] (Modelling and Analysis of Real-Time and Embedded systems profile) specialisation, that will be useful to complement the dependability properties with performance ones. On the other hand, since we focus our work in the context of intrusion-tolerant systems (i.e., those critical systems which apply FT techniques to tolerate intrusions), this implies also the necessity to report security requirements in the same UML designs. So, to avoid greater complexities, we rely on SecAM [8] (Security Analysis and Modelling profile), which is properly integrated in the MARTE-DAM framework. Although it may seem that the use of these profiles may bring some knottiness, in reality, a little part of the stereotypes proposed by the above mentioned profiles greatly helps designers of critical systems in their work.

The balance of the paper is as follows. Section 2 introduces the basis of the paper, i.e., some FT techniques we will use to illustrate the proposal and UML profiles. Section 3 presents UML and formal models for these FT techniques. Section 4 describes the UML design of an example and illustrates how the FT techniques can supplement it, moreover it shows how to obtain a final system formal model. Finally, related work and some conclusions are given in Section 5.

2 Previous Concepts

Before starting the contribution, we summarise in Section 2.1 the kind of FT techniques used along this work and in Section 2.2 we recall our proposal of a security profile in the context of MARTE-DAM, i.e., SecAM [8].

2.1 Proactive and reactive techniques

Modern critical systems (e.g., CRUTIAL [9]) incorporate fault prevention and fault-tolerant techniques to get a more robust system protected against faults.

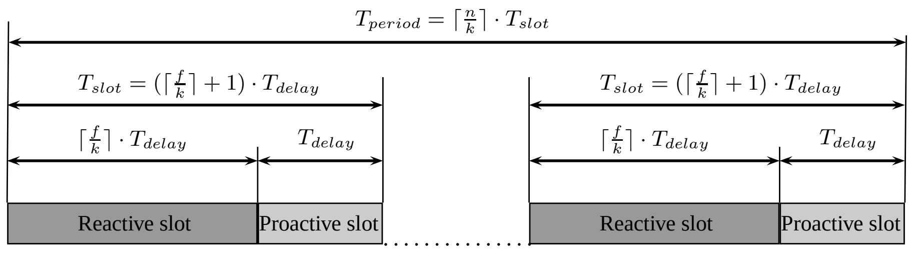
They are known as *intrusion-tolerant systems* when protection mainly concerns with faults coming from intrusions. Fault-tolerant techniques can be subdivided [1] in several groups: fault detection, fault recovery, fault handling and fault masking. In this work, we focus on proactive and reactive fault-tolerant recovery techniques.

An extensive review of the application of fault-tolerant techniques to the security domain can be found in [10]. It is also worth mentioning the work in [11], which suggests an approach to the design of highly secure computing systems based on fault-tolerant techniques. An interesting example of application can be found in [12], where the authors propose a technique for fragmenting the data and then storing the pieces in different locations (as RAID technology actually works), which reduces losses of data in case of intrusion.

Proactive recovery transforms a system state containing one or more errors (or even visible faults) into a state without detected errors or faults. Proactive techniques were presented in [13] as a long-term protection against break-ins and implemented, for example, in the scope of an on-line certification-authority system [14]. These techniques borrow ideas from proactive discovery protocols (e.g., IPSec [15]), session-key refreshment (SSL/TLS [16]) or secret sharing algorithms [17]. Hence, proactive security is defined as a combination of periodic refreshment and distribution [18,19].

Following Avizienis et al.'s fault taxonomy [1], *reactive recovery* can be classified as a fault-tolerant technique: it does a concurrent error detection, that is, errors in the system are detected meanwhile it is working. Then, a detection implies some actions must be performed in order to recover the system to a free-error state.

Proactive and reactive recovery techniques should not be considered as mutually exclusive but as complementary. Briefly, proactive techniques are worried about fault prevention (passive part of the system), while reactive ones are concerned with fault removal (active part). Sousa et al. presented in [20] a real application of proactive and reactive recovery techniques to an existing critical system, which tolerates up to f failure nodes and is able to recover in parallel up to k nodes. The rationale behind this idea is a scheduled time-line, which will be modelled in Section 3 (Fig. 1, adapted from [20], depicts it) and it is explained in the following.



$T_{period} = \lceil \frac{n}{k} \rceil \cdot T_{slot}$
$T_{slot} = (\lceil \frac{f}{k} \rceil + 1) \cdot T_{delay}$
$\lceil \frac{f}{k} \rceil \cdot T_{delay}$
T_{delay}
Reactive slot
Proactive slot
$T_{slot} = (\lceil \frac{f}{k} \rceil + 1) \cdot T_{delay}$
$\lceil \frac{f}{k} \rceil \cdot T_{delay}$
T_{delay}
Reactive slot
Proactive slot

Fig. 1. Schedule time-line showing activations of reactive and proactive recoveries.

A system with n distributed devices is initially divided into $\lceil \frac{n}{k} \rceil$ groups, containing each one up to k devices, being k the number of simultaneous recoveries the system can support. Assuming a period of time T_{period}, then each one is divided in $\lceil \frac{n}{k} \rceil$ slices (called T_{slot} from now on) where both (i.e., proactive and reactive) recoveries have to be performed. In a T_{slot}, one proactive recovery will be activated for a selected group which has a duration equal to T_{delay}, being T_{delay} the maximum expected time for recovering a device. Regarding reactive recovery, if we assume up to f failures and k simultaneously recoveries, that implies a maximum of $\lceil \frac{f}{k} \rceil$ reactive activations may happen in a T_{slot}. As can be inferred, T_{slot} has a duration equal to $(\lceil \frac{f}{k} \rceil + 1) \cdot T_{delay}$. There exists a relation [20] between values of n, f and k as is shown in Equation 1.

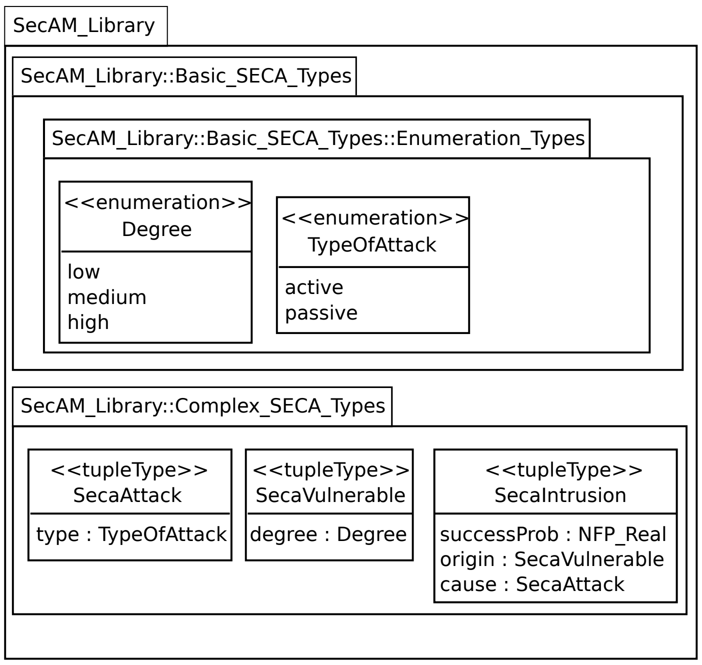
$$n \geq 2 \cdot f + k + 1 \tag{1}$$

A deeper description of the schedule time-line for proactive and reactive recoveries, as well as justification for inequality shown in Equation 1, can be found in [21].

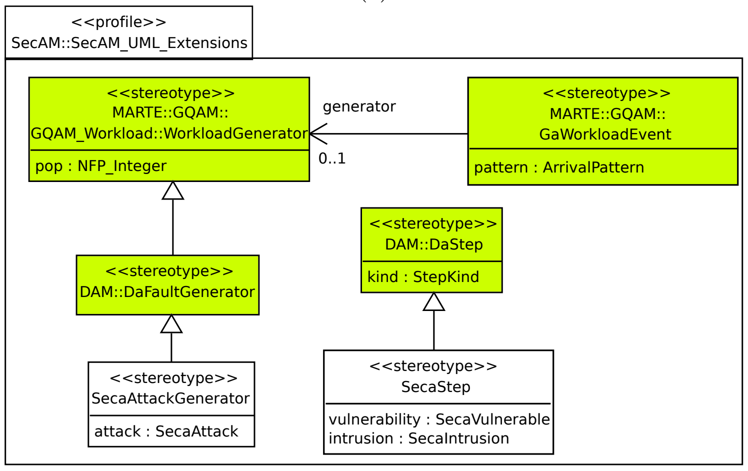
2.2 The Security Analysis and Modelling (SecAM) profile

The UML [2] (Unified Modelling Language) is a standard and comprehensive language that allows to specify functional software requirements through diagrams from architectural to deployment system views. UML can be tailored for analysis purposes through profiling. A *profile* defines stereotypes and tagged values for annotating design model elements extending its semantic. In particular, the Modelling and Analysis of Real-Time and Embedded systems (MARTE) [7] profile enables UML to support schedulability and performance analysis for real-time (RT) and embedded systems. Although focussed on RT, MARTE sub-profiles for performance and schedulability have also been proved useful in a wide range of other application domains. Performance as a Non-Functional Property (NFP) is specified in the MARTE context according to a well-defined Value Specification Language (VSL) syntax. Recently, the non-standard Dependability Analysis and Modelling (DAM) [6] profile was introduced to address dependability also as a NFP in UML design models. Indeed, as DAM is a MARTE specialisation, they can play together to specify performance and dependability NFPs in UML models. The entire set of MARTE stereotypes can be found in [7], while DAM stereotypes, including UML meta-classes that the stereotypes can be applied to, can be found in [6].

The close relation among dependability and security, cleverly disclosed by Avizienis et al. [1], was an argument in [8] for developing a new profile, called Security Analysis and Modelling (SecAM), to model and analyse security NFPs. Currently, the SecAM profile only addresses the topic of resilience, although its design favours easy integration of other security concerns. As the SecAM profile was constructed on top of DAM (indeed, as its specialisation), a joint DAM-SecAM specification on a UML design allows to accomplish a comprehensive dependability and security specification of system NFPs. The work here presented relies on the DAM-SecAM relation: we aim to specify fault-tolerant techniques, a dependability issue, for intrusion-tolerant systems, a security issue.


SecAM_Library
SecAM_Library::Basic_SECA_Types
SecAM_Library::Basic_SECA_Types::Enumeration_Types
<<enumeration>> Degree
low
medium
high
<<enumeration>> TypeOfAttack
active
passive
SecAM_Library::Complex_SECA_Types
<<tupleType>> SecaAttack
type : TypeOfAttack
<<tupleType>> SecaVulnerable
degree : Degree
<<tupleType>> SecaIntrusion
successProb : NFP_Real
origin : SecaVulnerable
cause : SecaAttack

(a)


<<profile>> SecAM::SecAM_UML_Extensions
<<stereotype>> MARTE::GQAM:: GQAM_Workload::WorkloadGenerator
pop : NFP_Integer
generator
0..1
<<stereotype>> MARTE::GQAM:: GaWorkloadEvent
pattern : ArrivalPattern
<<stereotype>> DAM::DaFaultGenerator
<<stereotype>> DAM::DaStep
kind : StepKind
<<stereotype>> SecaAttackGenerator
attack : SecaAttack
<<stereotype>> SecaStep
vulnerability : SecaVulnerable
intrusion : SecaIntrusion

(b)

Fig. 2. (a) SecAM library and (b) SecAM UML extensions.

Figure 2(b) depicts some SecAM stereotypes used in this work, concretely `secaStep` and `secaAttackGenerator`. The SecAM library (Fig. 2(a)) describes the types associated to the tagged values of these stereotypes. The `secaStep` stereotype inherits from `DAM::daStep` stereotype and is meant to describe a system vulnerability or an attack, being both security faults [8]. For description of system malicious intrusions, the `secaAttackGenerator` stereotype is introduced, besides, the MARTE and DAM classes it specialises (Fig. 2(b)) allow to describe the occurrence probability pattern of the intrusion. `DaFault` DAM annotation, later used in this work, supports fault definition in [1] and means the basis for the actual SecAM annotations.

3 Modelling Proactive and Reactive Recovery Techniques

We develop, in this section, a generic and reusable model of proactive and reactive recovery techniques. In first term, we model them using UML state-machine (UML-SC) diagrams annotated with the previously discussed profiles. Then, we obtain a Coloured Petri Net (CPN) [22] which maps the behaviour of these UML diagrams. In fact, this CPN accurately represents proactive and reactive recovery techniques. Our intention is then to reuse such CPN through different software designs to conclude about the appropriateness of the techniques for the design, Section 4 will show an example. To accomplish this target, these software designs will also be modelled using UML-SC and each one will be eventually converted into a CPN. So, our proposal to reuse the "proactive-reactive" CPN within a given software design has to offer adequate "interfaces" to compose both CPNs. Then, we finally get a CPN that embeds both the proactive-reactive techniques and the software design as explained in Section 4.

3.1 UML Modelling

We have distinguished two components, one in charge of controlling the scheduled time-line presented in Section 2, and the other controlling the device to be recovered. The latter has been called Proactive and Reactive Recovery (PRR) component following terminology in [20].

Schedule controller UML state-chart is depicted in Figure 3. Initial analysis variables (`gaAnalysisContext` stereotype) are: `tDelay`, which determines the duration of each recovery; `f`, number of faulty devices allowed; and `k`, number of devices recovered in parallel. Only one controller will be placed in the system (tag `pop` of `gaWorkloadGenerator` stereotype). Once created, it calculates in `g` the first group which will be proactively recovered. Upon entrance into *Reactive slot* state, it invokes event `nextSchedule()` for PRR devices in `g` to inform them that the components they control will be proactively recovered in the next proactive slot, so their monitoring activity will not be necessary since for sure they will be recovered. Then, it starts the `countDown()` activity with duration `hostDemand` equals to $\lceil \frac{f}{k} \rceil \cdot tDelay$ seconds (that is, it makes room for up to $\lceil \frac{f}{k} \rceil$ parallel recoveries). Completion of `countDown()` activity means to schedule

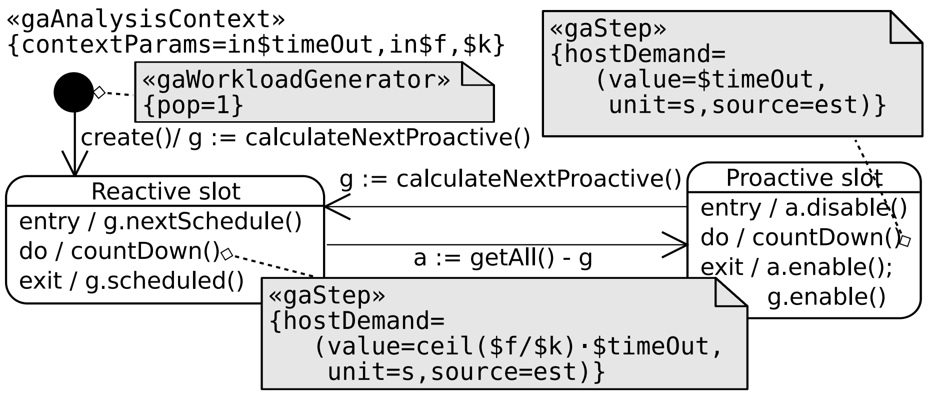


Fig. 3. Scheduler UML state-machine diagram.

elements in g and to change to *Proactive slot* state, where all PRR devices are disabled and it starts the proactive `countDown()` activity, in this case with a duration equal to `tDelay` seconds. Once finished, it enables all PRR devices and before entering again in the *Reactive slot* state, it calculates the next proactive group.

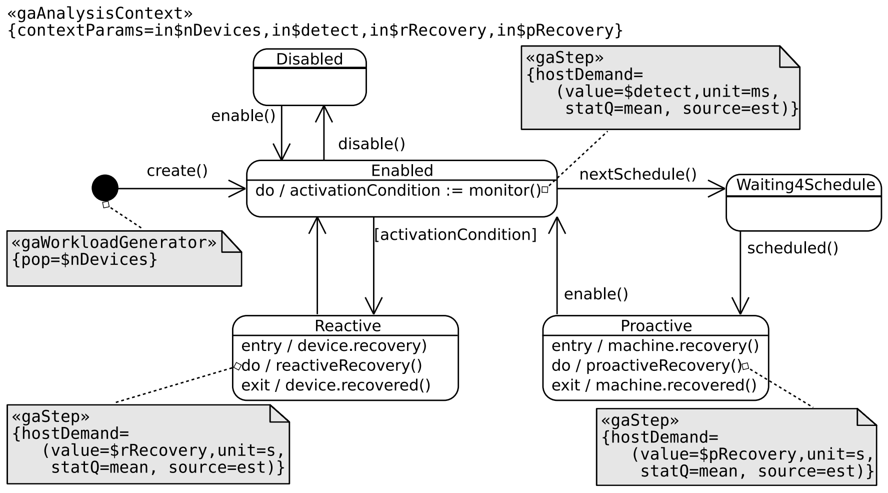


Fig. 4. PRR controller UML state-machine diagram.

Figure 4 shows UML-SM for PRR component controller. Obviously, the population is equal to the number of effectively monitored devices, `nDevices`. It starts in *Enabled* state and executing the activity `monitor()`, which abstracts two processes: 1) detection of errors in the monitored device and 2) checking for room in current time slot for a reactive recovery. So, when it positively informs, then enters in *Reactive* state to perform a recovery (`reactiveRecovery()` activity), which has a duration of `rRecovery` seconds on average. Once finished, it comes back to *Enabled* state. From there, event `nextSchedule()` evolves to *Waiting4Schedule* state, where the PRR will wait for event `scheduled()` invoked by the scheduler to start the proactive recovery. In both recovery states (*Reactive* or *Proactive*) the PRR invokes upon the entrance (`recovery()`) and on the exit (`recovered()`) events in the monitored device switching it off/on, respectively. Finally, note that events `enable()` and `disable()` received from the scheduler effectively prevent the PRR to monitor its device.

Token colour definitions
$type\ D\ is\ \{1 \ldots nDevices\}$
$type\ G\ is\ \{G_1 \ldots G_{\lceil \frac{nDevices}{k} \rceil}\}$
$subtype\ G_i\ is\ \{(k \cdot (i-1)+1) \ldots k \cdot i\}$
$var\ i:\ D,\ g:\ G$

Initial marking
$m_0(Enable) = \sum i \in D$
$m_0(nextGroup) = G_1$
$m_0(Idle) = 1$
$m_0(maxParallel) = k$

Functions definitions
$belonging(g:\ G) = \sum i \in G$
$cSubset(g:\ G) = \sum i \in D \vert i \ni G$
$allDevices() = \sum i \in D$

Table 1. CPN initial marking, token colour definition and functions.

3.2 Formal Modelling through Petri Nets

Following ideas in [5] we obtained two Generalized Stochastic Petri Nets (GSPNs) [23] by model transformation of UML design (Figs. 3 and 4). Considering that ideas proposed in [5] were given for performance analysis purposes, some minor changes have emerged. Indeed, we used the ArgoSPE [24] tool, which implements the algorithm given in [5], to perform the transformation of UML-SCs annotated with SPT [25] (Schedulability, Performance and Time profile, precursor of MARTE) into GSPNs. Seeing that ArgoSPE does not support MARTE, nor DAM nor SecAM profiles, the GSPNs obtained from the transformation have been manually modified to incorporate such annotations. In the following we summarise the algorithm implemented in ArgoSPE.

Each SC simple state is transformed in a PN place, which represents its entrance. The latter is followed by two causally connected PN transitions which represent, respectively, the entry action and the do-activity of the SC state. Entry actions are modelled by immediate PN transitions (assuming its execution time is negligible) while do-activities are represented by timed PN transitions, which are characterised by one output place (i.e., the completion place) modelling the

SC completion state. If the SC state has outgoing immediate transition, this is translated into a PN immediate transition with the completion place as its input place. For conflicting outgoing transitions, the transformation adds immediate transitions with probabilities to stochastically resolve the choice. In this case, the probability values are taken from the annotations attached to the transitions.

In order to take advantage of the hierarchy and symmetries in the problem, and then gaining readability, we slightly modified these semi-automatically obtained Petri nets to gain an equivalent Coloured Petri Net (CPN) [22]. Figure 5(b,c) depicts these CPNs, while Figure 5(a) offers a hierarchical CPN [26] view for an easy understanding of interactions between each subnet. Interactions occur via event places (*e_enable*, *e_nextSchedule*, *e_schedule* and *e_disable*), which model the real interfaces among components. Controlled devices are also depicted (light gray box) just to highlight their communication with the PRR controllers. CPNs in Figure 5(b,c) depict deterministic delays through grey transitions, stochastic delays through white transitions, while black transitions are immediate ones. The initial marking, token colour and functions definition are summarised in Table 1.

The CPN of scheduler SC (Figure 5(c)) has an initial marking represented in places `Idle` and `nextGroup`, with values indicated in Table 1, which come from `gaWorkloadGenerator` annotation and `calculateNextProactive()` function in the SM. Transitions `fkTOut` and `TOut` represent the reactive and proactive countdowns and are characterised by deterministic durations given in the corresponding `gaStep` SC annotations. Note that the firing of `TOut` enables all PRR controllers (through place `e_enable`), generates the next group which will be proactively recovered and starts up the cycle again.

Regarding CPN of PRR controller (Figure 5(b)), its initial marking in places `Enabled` and `maxParallel` represent the number of PRR devices in the system and the maximum number of devices the system can recover in parallel, according to annotations in the SC. Firing of transitions `PRactions` and `RRactions` respectively lead the activation of proactive and reactive recovery. Monitored devices are informed about the starting and ending of both recoveries (proactive and reactive) through places `e_recovery` and `e_recovered`. Transition `Detect` abstracts the activity `monitor()` in the SC, which once fired checks activation conditions to, in positive case, inform the device about the starting of the reactive recovery. Note that this can take place only if there exist room enough for a new parallel recovery.

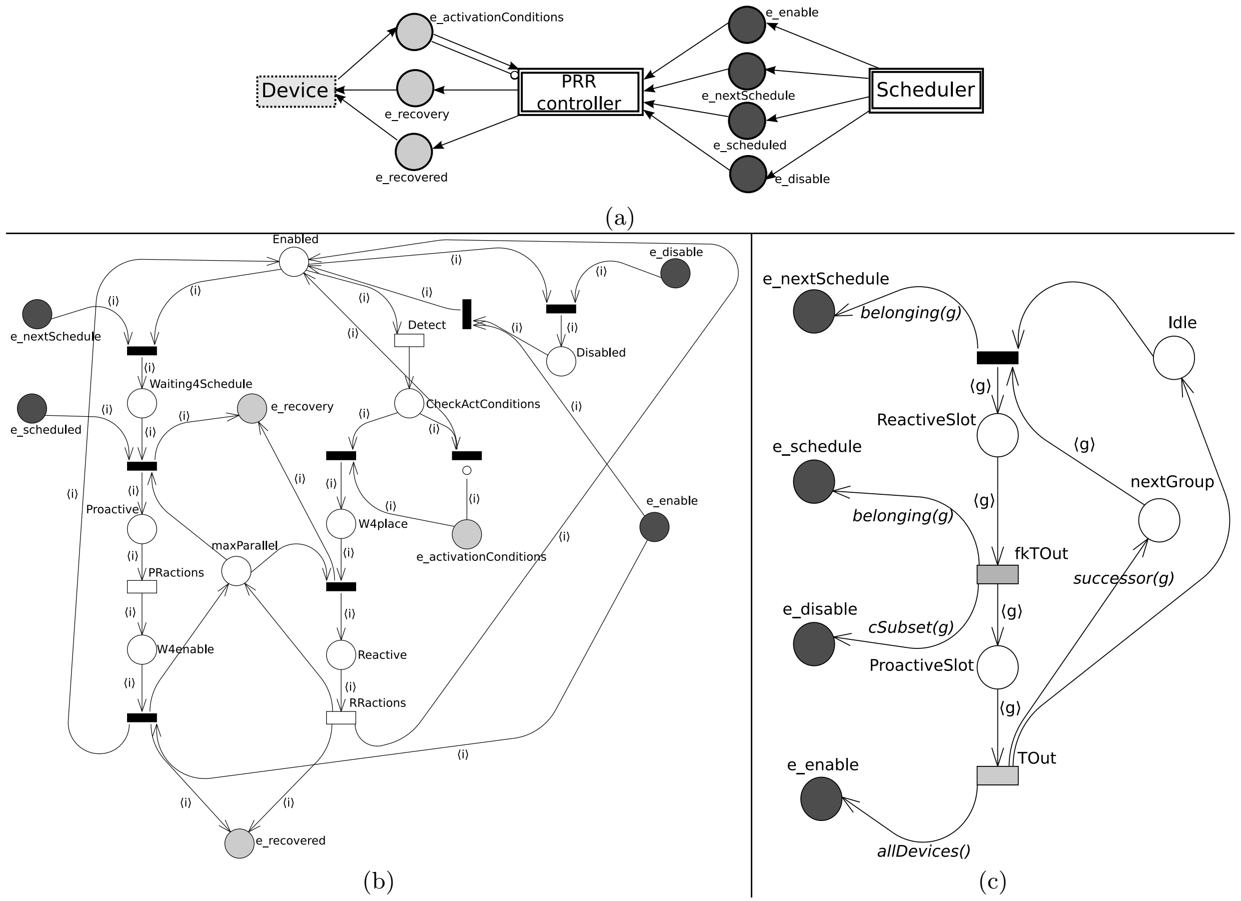


Fig. 5. (a) Hierarchical CPN, (b) CPN of PRR controller and (c) CPN of scheduler.

4 Example: On-line Shopping Website

Proactive and reactive recovery techniques such as the ones here described, but also many others, can be implemented in critical systems. However, it would be highly interesting to assess their actual convenience for a given system before to carry out them physically. Thereby, we think that the models developed in previous section can be useful for this purpose and with this thought in mind, we show in this section how a software design can offer interfaces through which eventually it will be combined, at UML level, with the FT target techniques. The example tries to be a blueprint about how to:

(a) add proactive and reactive techniques to a critical system for improving its fault tolerance;
(b) obtain an analysable formal model and
(c) get results from such model that can assess system dependability.

We model a business-critical system of the kind of an on-line shopping website, where a balance loader is in charge of placing customers in several servers. Each server manages a defined number of customers in parallel. A physical view of the system is depicted in Figure 6. Note that it incorporates an external PRR device (assumed tamper-proof and not subjected to failures) that embeds proactive and reactive recovery techniques. The system also features the scheduler device wired to PRRDs. Even achieving a complete failure prone device it is not a reality, this kind of device can be seen as an embedded tamper-proof device, that is, there is no possibility of deliberate altering or adulteration of the device. The addition of other FT techniques (e.g., replication, redundancy or diversity) to PRR devices will still fit within the techniques presented in this paper because the effort should be done in the modelling of the interaction between techniques.

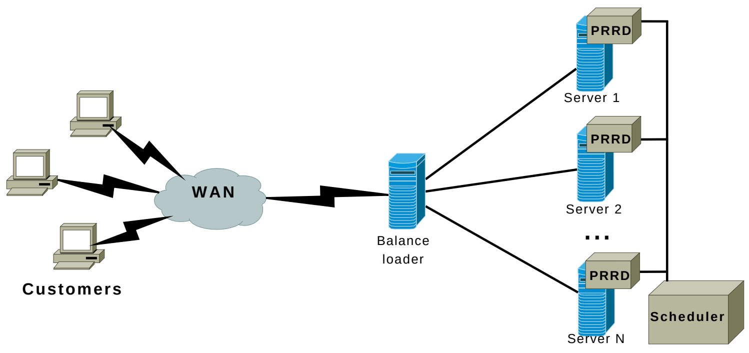


Fig. 6. Physical view of the system.

4.1 UML Modelling

Figure 7 depicts the behaviour of the balance loader, of which only one copy exists (`pop` tagged value) and starts in *Idle* state. An open workload generates customer's requests (`gaWorkloadGenerator` stereotype) with an inter-arrival time defined as an exponential distribution. Customer arrivals provoke this component to execute an algorithm (`balanceLoader()`) that ends up asking a server to attend the new customer. This component does not interplay with the target techniques, so no interface is required. However, its behaviour is mandatory to get some system parameters such as workload (see `gaWokloadEvent` annotation).

Server state-machine diagram is depicted in Figure 8. Initial pseudo-state indicates `nDevices` servers ready in the system. Each *available* server can concurrently attend up to `nThreads` started up through event `attendCustomer()`, which indeed initiates a sub-state machine specified in Figure 9. The server has been supplied with interfaces (`recovery()` and `recovered()`) to interact with the PRR component via events (note that here is where we incorporate the recovery techniques into the system). Consequently, the actual functional behaviour of a server is specified in the sub-state machine, which inherits the interfaces then allowing to abort normal behaviours. Besides, we have wanted to show how other kinds of faults, e.g. hardware faults, can also be expressed within this modelling approach. So, when a hardware crash occurs (`daStep` annotation in Fig. 8) it will be properly handled, of course the resulting formal model will also embed this kind of fault.

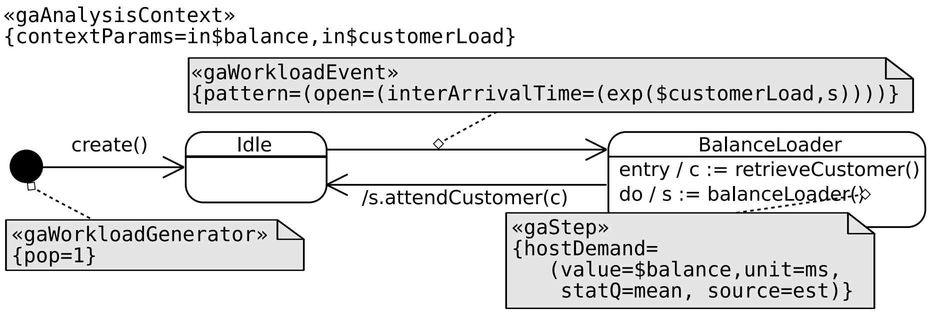


Fig. 7. Balance loader UML state-machine diagram.

During normal behaviour (Fig. 9) a server can be attacked and/or suffer intrusions. In the example we have reduced, as much as possible, the specification of system normal behaviour (*Processing*) to focus on the *critical* part. Hence, the customer's requests (`attendCustomer`) can be a source of attack (see `secaAttackGenerator` annotation) and occasionally become an intrusion (`secaStep` annotation), i.e., the attack successes. Obviously, other kind of dependability faults could be here specified by means of DAM-SecAM. A final remark to point out that conflicting outgoing transitions of *Processing* state are evidently solved by the probabilities in the annotations.

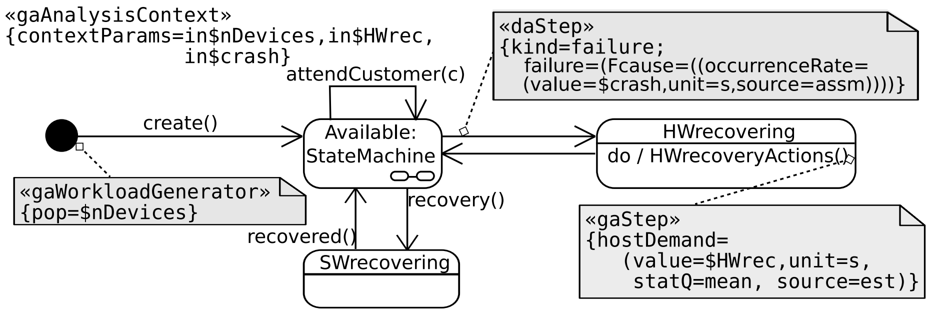


Fig. 8. Server UML state-machine diagram.

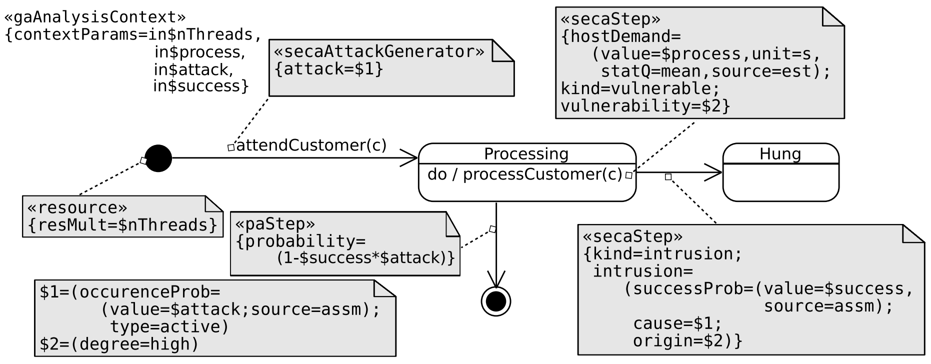


Fig. 9. Available UML sub-state machine diagram.

4.2 Formal Modelling

Again, following ideas in [5] and assisted by ArgoSPE [24] tool, we obtained GSPNs by model transformation of UML design (Figs. 7, 8 and 9). Thereafter, the nets were composed by interface places (`e_attendCustomer`), simplified and converted into a CPN (depicted in Fig. 10) for readability purposes. The initial marking and transition rates are summarised in Table 2.

Interface light grey places allow combination with the PRR component as depicted in Figure 5(a), so to gain the target Petri net that models both: system behaviour and recovery techniques. Now we can discuss the role of interface place `e_activationConditions`, but firstly remember that a token in this place means for the PRR component to activate a reactive recovery. From the point of view of our system, we desire to activate the reactive recovery whenever 7 out of 10 threads (see *nThreads* variable in Table 2(b)) in a given server become hung. So, the `actConditions()` function in the test arc of place *ThreadHung* implements the algorithm that checks out such condition and when it is true then a token is placed in place `e_activationConditions`. Obviously, other systems should implement this function differently, but always preserving place `e_activationConditions` as an interface place.

In *Balance loader* area, the system open workload is represented by an exponentially distributed transition `customerArrival` of mean *customerLoad* (taken from `gaWorkloadEvent` stereotype, Fig. 7). In the *Thread* area, transitions `intrusion` and `nonintrusion` represent whether if an attack had success or not, respectively. Finally, we manually added the part of the net called *Cleaning* (indeed, composed only by two transitions) to remove tokens from `Processing` and `ThreadHung` as long as the server becomes not available (arc inscription $\#P_i$ means all tokens in place P_i).

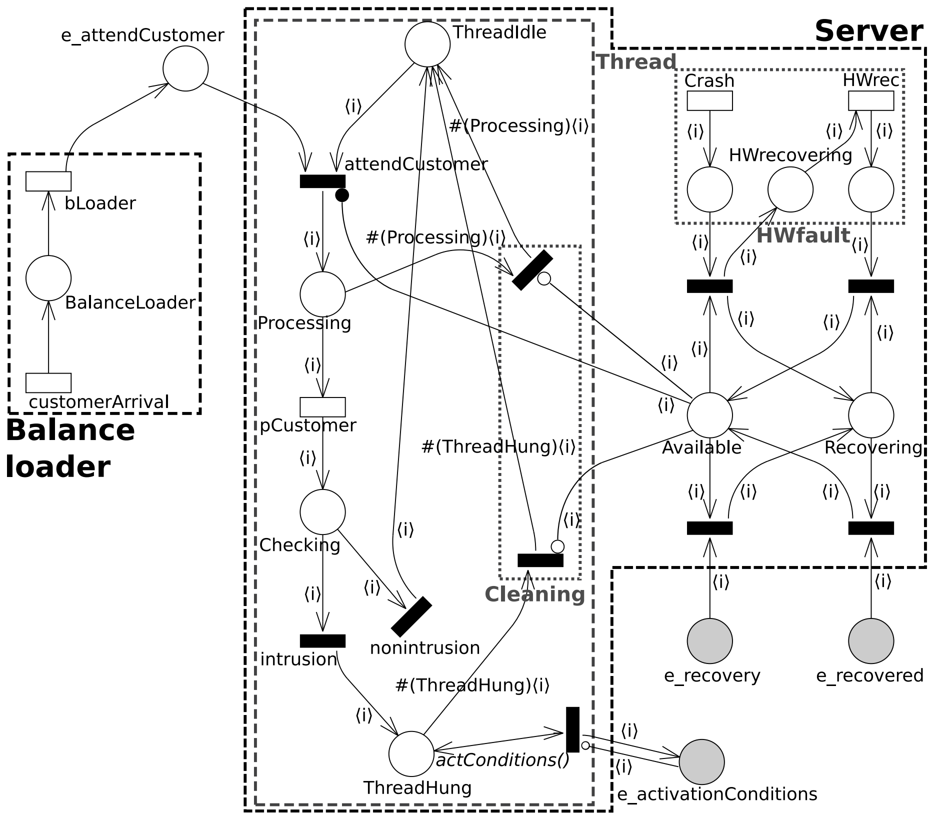


Fig. 10. CPN of case study.

Initial marking
$m_0(ThreadIdle) = nThreads \cdot \sum i \in D$
$m_0(Idle) = \sum i \in D$

Transition	Parameter (type)
customerArrival	$1/customerLoad$ (rate)
bLoader	$1/balance$ (rate)
pCustomer	$1/process$ (rate)
Crash	$1/crash$ (rate)
HWrec	$1/HWrec$ (rate)
intrusion	$attack \cdot success$ (weight)
nonintrusion	$1 - attack \cdot success$ (weight)

(a)

Parameters	Value
$nDevices$	12
k	$2, 3, 4$
f	1
$timeOut$	$120, 180s$
$detect$	100 ms
$pRecovery$	120 s
$rRecovery$	120 s
$nThreads$	10
$crash$	432000 s
$HWrec$	43200 s
$balance$	200 ms
$customerLoad$	0.5 customers/s
$process$	300 s
$attack$	30%
$success$	$0\% \cdots 75\%$

(b)

Table 2. (a) Example parameters and (b) experiments parameters.

4.3 Analysis and Assessment

The analysis was carried out using simulation programs of GreatSPN [27] tool. We actually simulated the original GSPNs obtained by ArgoSPE instead of the readable CPNs in Figures 5 and 10. Simulation parameters were set to a confidence level of 99%, accuracy of 1%, length of evolution phase equal to 604800 time units and a length of initialisation phase equal to 86400 time units. The Petri nets parameters and its values were summarised in Table 2(b) (note that the number of servers was 12 (`nDevices`), each one able to attend up to 10 customers in parallel (`nThreads`)).

All values of the parameters can be known at design time: some of them, such expected customer load, probabilities of attack and success, should be estimated by the software engineer, while other parameters, such as time performing recovery actions should be given by manufacturer of PRR device. Table 3 summarises input analysis parameters and by whom they should be provided.

In the experiments, reactive recovery is always performed when the number of active threads for a device drops to 3 (function `activationConditions()`). Regarding proactive recovery, 12 servers allow setting several configurations that we have tested: *three proactive groups* of four servers (solid lines in Figure 11), *four proactive groups* of three servers (dot-dashed lines) and *six proactive groups* of two servers (dashed lines). Under these parameters, we have simulated the net to point out the best configuration among the previous ones w.r.t. throughput. In Figure 11 we show the relation between the incoming customer throughput (`customerLoad`) and the system throughput (*Thr(attendCustomer)*). The horizontal axis represents the percentage of successful attacks (0%..75%, variable

Input parameter	Provided by	Annotation (profile)
`balance`	Manufacturer	`gaWorkloadEvent` (MARTE)
`customerLoad`	Designer	`gaStep` (MARTE)
`nDevices`	Designer	`gaWorkloadGenerator` (MARTE)
`HWrec`	Manufacturer	`gaStep`
`crash`	Manufacturer	`daStep` (DAM)
`nThreads`	Manufacturer	`resource` (MARTE)
`process`	Manufacturer	`secaStep` (SecAM)
`attack`	Designer	`secaAttackGenerator` (SecAM), `paStep` (MARTE)
`success`	Designer	`secaStep` (SecAM), `paStep` (MARTE)

Table 3. Analysis parameters required in UML statecharts.

`success`), having the percentage of system attacks (variable `attack`) set to 30% for all the experiments.

The results indicate that the more servers are simultaneously recovered, the more throughput the system obtains. In terms of absolute time, smaller groups recover more number of servers than bigger groups, which ensures higher availability for the formers and consequently better performance. In the example, it could be assessed that groups of four serves are the right choice regarding throughput. The computed measure is a *performability* one (i.e., performance in the presence of faults). Although some other interesting results were obtained (e.g., results of dependability nature) from this formal model, we do not present them since this is not the main focus of the paper.

5 Related Work and Conclusion

Several approaches [28,29,30] in the literature bring Petri nets for the design of critical systems. In [28] Heiner et al. used a combined model of Z and Petri Net formalisms, the first for specifying data and its evolution and the latter to validate the safety-critical system. The union of both formalisms allows to obtain an approach where data-combination and behaviour are described. Ghezzi et al. presented in [29] a high-level Petri Net formalism (TB nets, a particular case of *Timed Environment/Relationship* nets) to specify control, function and timing issues in time-critical systems. In [30], Houmb et al. quantified operational system integrity of security critical systems using the formalism of Coloured Petri Nets (CPN).

Regarding fault-tolerant techniques applied at software architectural level also several works can be found [31,32,33,4,34]. In [31] Harrison and Avgeriou studied how several fault-tolerant techniques can be carried out as best-known architectural patterns. By the use of architectural patterns they aim to directly create software architectures satisfying quality attributes. Nguyen-Tuong and

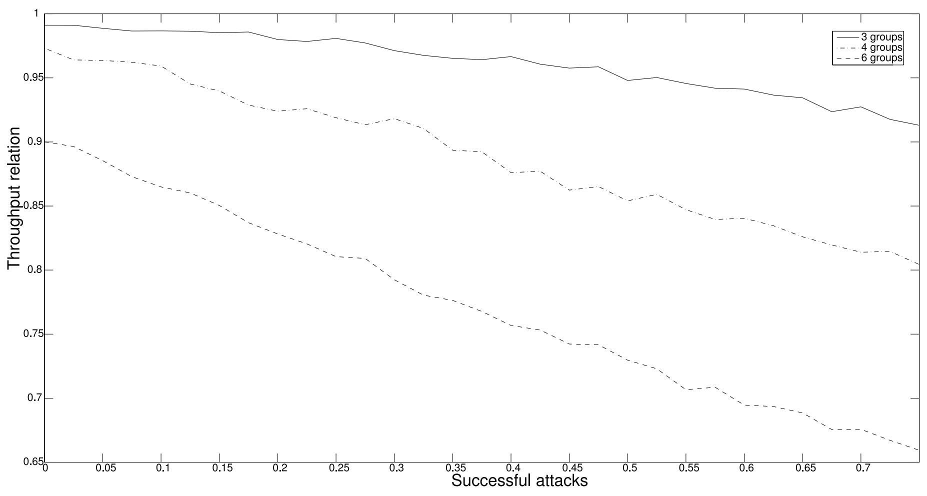


Fig. 11. Simulation results.

Grimshaw presented in [32] a reflective model, called *Reflective Graph & Event* (RGE), which is applied for making failure-resistant applications. Using this reflective model they are able to express fault-tolerant algorithms as reusable components allowing composition with user applications. Rugina et al. propose in [33] an approach for system dependability modelling using AADL (Architecture Analysis and Design Language), being the design model transformed into GSPN. This approach was applied to an Air Traffic Control System. Bondavalli et al. [4,34] have a vast work in the area of translating UML diagrams into dependability models, having also used Petri nets as a target model in some of these works. Their proposal of translation could be used in this paper instead of [5], but it should be taken into account that they propose an intermediate model as a first step.

In this paper, we have explored the idea of combining models that represent FT techniques and software behavioural designs. The combined model is useful for dependability assessment. Although the example has shown feasibility in the approach to integrate well-known recovery techniques into software designs, we are conscious that a long path has to be walked for the approach to reach applicability. So, we want to clearly establish that, from our point of view, the contribution of the paper is restricted to the achievements in the example, i.e., how to combine proactive and reactive techniques with a software design and their analysis. However, we are confident of the second one, i.e., reuse of the approach with other FT techniques. The key point is to gain a "library" of UML models representing FT techniques ready to use in critical designs. Being the crucial aspect for the UML model of a FT technique to have clearly defined its interfaces, we strongly believe that *events* and *conditions* are the means to attain it as we did in our proposal. Moreover, each technique has to define also how their interfaces play in the software design, for the case of the recovery techniques we have advocated for a superstate which offers suitable interfaces and embeds the system normal behaviour.

As a critic, we recognise that UML-SC should not be the only UML diagram used in this context, since for example, sequence diagrams may help the engineer in understanding system usage situations. So, we plan to extend our approach to take advantage of other UML diagrams. Another critic stems from the fact that the combination of FT techniques and software designs should be explicitly made at UML level, instead of deferring the combination to the Petri net models. This would bring advantage to the engineer for completely avoid the formal model. Being aware of this fact, we are working on a feasible solution to this problem.

The use of an approach such as the one here developed should otherwise bring several benefits from the point of view of a software engineer. The easy integration of FT techniques into software designs and the existence of such "library" may allow to test different techniques for the same design to find the ones fitting better. Such "library" will also free the engineer of worrying about how to model FT and concentrate on the problem domain. Finally, it is well-known that the use of formal models early in the life-cycle to prove requirements is less expensive than other approaches.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. on Dependable and Secure Computing **1**(1) (2004) 11–33
2. OMG: Unified Modelling Language: Superstructure. Object Management Group. (July 2005) Version 2.0, formal/05-07-04.
3. Murata, T.: Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE. Volume 77. (April 1989) 541–580
4. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability Analysis in the Early Phases of UML Based System Design. Journal of Computer Systems Science and Engineering **16**(5) (2001) 265–275
5. Merseguer, J., Bernardi, S., Campos, J., Donatelli, S.: A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In Giua, A., Silva, M., eds.: Procs. of the 6th Int. Workshop on Discrete Event Systems, Zaragoza, Spain, IEEE Computer Society Press (October 2002) 295–302
6. Bernardi, S., Merseguer, J., Petriu, D.: A Dependability Profile within MARTE. Journal of Software and Systems Modeling (2009) DOI: 10.1007/s10270-009-0128-1.
7. Object Management Group: A UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE). (November 2009) v1.0, formal/2009-11-02.
8. Rodríguez, R.J., Merseguer, J., Bernardi, S.: Modelling and Analysing Resilience as a Security Issue within UML. In: SERENE'10: Procs. of the 2nd Int. Workshop on Software Engineering for Resilient Systems, ACM (2010) Accepted for publication.
9. Veríssimo, P., Neves, N.F., Correia, M., Deswarte, Y., Kalam, A.A.E., Bondavalli, A., Daidone, A.: The CRUTIAL Architecture for Critical Information Infrastructures. In de Lemos, R., Giandomenico, F.D., Gacek, C., Muccini, H., Vieira, M., eds.: WADS. Volume 5135 of LNCS., Springer (2007) 1–27

10. Rushby, J.: Critical System Properties: Survey and Taxonomy. Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International (1994)
11. Dobson, J., Randell, B.: Building Reliable Secure Computing Systems Out Of Unreliable Insecure Components. In: IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA, IEEE Computer Society (1986) 187
12. Fray, J.M., Deswarte, Y., Powell, D.: Intrusion-Tolerance Using Fine-Grain Fragmentation-Scattering. In: IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA, IEEE Computer Society (1986) 194
13. Canetti, R., Gennaro, R., Herzberg, A., Naor, D.: Proactive Security: Long-term Protection Against Break-ins. CryptoBytes **3** (1997) 1–8
14. Zhou, L., Schneider, F.B., Van Renesse, R.: COCA: a Secure Distributed Online Certification Authority. ACM Trans. on Computer Systems (TOCS) **20**(4) (2002) 329–368
15. Tran, T.: Proactive Multicast-Based IPSEC Discovery Protocol and Multicast Extension. MILCOM **0** (2006) 1–7
16. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force (April 2006)
17. Shamir, A.: How to Share a Secret. Communications of ACM **22**(11) (1979) 612–613
18. Canetti, R., Halevi, S., Herzberg, A.: Maintaining Authenticated Communication in the Presence of Break-ins. In: PODC '97: Procs. of the 16th annual ACM symposium on Principles Of Distributed Computing, New York, NY, USA, ACM (1997) 15–24
19. Ostrovsky, R., Yung, M.: How To Withstand Mobile Virus Attacks. In: PODC '91: Procs. of the 10th annual ACM symposium on Principles Of Distributed Computing, New York, NY, USA, ACM (1991) 51–59
20. Sousa, P., Bessani, A., Correia, M., Neves, N., Verissimo, P.: Resilient Intrusion Tolerance through Proactive and Reactive Recovery. Procs. of the 13th IEEE Pacific Rim Dependable Computing Conference (2007) 373–380
21. Kalan, A.A.E., Baina, A., Beitollahi, H., Bessani, A., Bondavalli, A., Correia, M., Daidone, A., Deconinck, G., Deswarte, Y., Garrone, F., Grandoni, F., Moniz, H., Neves, N., Rigole, T., Sousa, P., Verissimo, P.: D10: Preliminary Specification of Services and Protocols. Project deliverable, CRUTIAL: Critical Utility Infrastructural Resilience (2008)
22. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Monographs in Theoretical Computer Science. Springer-Verlag (1997)
23. Chiola, G., Marsan, M.A., Balbo, G., Conte, G.: Generalized Stochastic Petri Nets: A Definition at the Net Level and its Implications. IEEE Trans. Soft. Eng. **19**(2) (1993) 89–107
24. ArgoSPE: `http://argospe.tigris.org`
25. Object Management Group: UML Profile for Schedulability, Performance and Time Specification. (January 2005) V1.1, f/05-01-02.
26. Huber, P., Jensen, K., Shapiro, R.M.: Hierarchies in Coloured Petri Nets. In: APN 90: Procs. on Advances in Petri nets 1990, New York, NY, USA, Springer-Verlag New York, Inc. (1991) 313–341
27. University of Torino: The GreatSPN tool `http://www.di.unitorino.it/~greatspn` (2002)
28. Heiner, M., Heisel, M.: Modeling Safety-Critical Systems with Z and Petri Nets. In: SAFECOMP '99: Procs. of the 18th Int. Conf. on Computer Computer Safety, Reliability and Security, London, UK, Springer-Verlag (1999) 361–374

29. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A Unified High-Level Petri Net Formalism for Time-Critical Systems. IEEE Trans. Softw. Eng. **17**(2) (1991) 160–172
30. Houmb, S.H., Sallhammar, K.: Modelling System Integrity of a Security Critical System Using Colored Petri Nets. In: Proceedings of Safety and Security Engineering (SAFE 2005), Rome, Italy, WIT Press (2005) 3–12
31. Harrison, N.B., Avgeriou, P.: Incorporating Fault Tolerance Tactics in Software Architecture Patterns. In: Procs. of the 2008 RISE/EFTS Joint Int. Workshop on Software Engineering for Resilient Systems (SERENE), ACM (2008) 9–18
32. Nguyen-Tuong, A., Grimshaw, A.S.: Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications. Technical report, University of Virginia, Charlottesville, VA, USA (1998)
33. Rugina, A.E., Kanoun, K., Kaâniche, M.: A System Dependability Modeling Framework Using AADL and GSPNs. In de Lemos, R., Gacek, C., Romanovsky, A.B., eds.: WADS. Volume 4615 of LNCS., Springer (2006) 14–38
34. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic Dependability Analysis of System Architecture Based on UML Models. In De Lemos, R., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems. Volume 2677 of LNCS. Springer-Verlag, Berlin, Heidelberg, New York (2003) 219–244