Logical Specification and Analysis of Fault Tolerant Systems Through Partial Model Checking

S. Gnesi, G. Lenzini [2,1]

Istituto di Scienze e Tecnologie Informatiche - C.N.R.
Via G. Moruzzi 1, I-56100 Pisa, Italy

F. Martinelli [3]

Istituto di Informatica e Telematica - C.N.R.
Via G. Moruzzi 1, I-56100 Pisa, Italy

Abstract

This paper presents a framework for a logical characterization of fault tolerance and its formal analysis based on partial model checking techniques. The framework requires a fault tolerant system to be modeled using a formal calculus, here the CCS process algebra. To this aim we propose a uniform modeling scheme in which to specify a formal model of the system, its failing behaviour and possibly its fault-recovering procedures. Once a formal model is provided into our scheme, fault tolerance - with respect to a given property - can be formalized as an equational μ-calculus formula. This formula expresses, in a logic formalism, all the fault scenarios satisfying that fault tolerance property. Such a characterization understands the analysis of fault tolerance as a form of analysis of open systems and, thank to partial model checking strategies, it can be made independent from any particular fault assumption. Moreover this logical characterization makes possible the fault-tolerance verification problem be expressed as a general μ-calculus validation problem, for solving which many theorem proof techniques and tools are available. We present several analysis methods showing the flexibility of our approach.

Key words: Fault Tolerant Systems, Formal Verification, Partial Model Checking.

[1] Gnesi and Lenzini are partially supported by the MIUR-CNR project SP4. Martinelli is supported by MIUR project "Tools, techniques and methodologies for the information society".

[2] Email: {stefania.gnesi, gabriele.lenzini}@isti.cnr.it

[3] Email: fabio.martinelli@iit.cnr.it

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Following a general direction pointing out that it exists a beneficial interactions between strategies of analysis in security and fault tolerance (*e.g.,* see [14,19,24]), this paper inquires about the analogy between fault tolerance and a particular strategy for computer security analysis *i.e., partial model checking* [3,17,18].

As in many approaches using formal methods in the specification and validation of fault tolerant system (*e.g.,* see [8,5]), this paper first requires a system, its failing behaviour (with respect to fault occurrences) and its fault-recovering procedures to be formally specified as a finite state term in some process algebras, for example the CCS [20]. Then this paper requires desired properties to be formalized in logical formalism, here a variant of the μ-calculus [6]. A property expressed in such a logic can be formally checked over a formal model of the system under analysis, which is typically expressed by means of a state machine or transitions system.

The validation framework we propose falls into the *open system* paradigm, where a system is considered acting in an *environment* able to interact with the system. Anyway, the fact of considering a "general environment" able to invoke, in any order, any action of the system's interface, usually brings unpleasant drawbacks. The first is the well-known state space explosion, and the second is that unrealistic situations may arise during the analysis [9].

As a solution in this paper we are interested to verify a system P in a well-characterized environment $F_{\mathcal{F}}$. This latter, by acting as a fault-injector, is able to interact with the system only through the specified set $\mathcal{F}$ of fault actions. Moreover, differently to the usual strategies requiring the fault-injector to be specified (*e.g.,* see [26]), we treat $F_{\mathcal{F}}$ as a unspecified component of the system. In this way we mean to check the reliability of a system model with respectively to any potential occurrences of faults.

Moreover by using *partial model checking* strategies [3], the fault tolerance analysis problem can be reduced to a validity problem in the μ-calculus. Briefly the idea is the following: proving that $(P \parallel F_{\mathcal{F}}) \setminus \mathcal{F}$ satisfies a fault tolerant property *e.g.,* a μ-calculus formula φ, is equivalent to prove that $F_{\mathcal{F}}$ satisfies a modified formula $\varphi /\!\!/ P$, where $/\!\!/ P$ is the partial evaluation for the parallel composition and restriction operators. In this way $\varphi /\!\!/ P$ characterizes exactly the scenarios of faults the system is resilient to. As a consequence, by considering the characteristic formulas $\varphi_{\mathcal{F}}$ of all the possible fault scenarios, checking if P is fault tolerant is equivalent to check the validity of $\varphi_{\mathcal{F}} \Rightarrow \varphi /\!\!/ P$.

Once such a logical characterization of fault tolerance is given, several analysis techniques may be adopted. Some of them may also lead to efficient analysis methods for certain properties. In particular, we identify a class of μ-calculus formulas for which the analysis is very efficient.

2 Modelling Fault-Tolerant Systems

This section presents a uniform scheme where a fault tolerant system may be specified as a process algebra term. To specify a system we will use the CCS [20], but our framework is completely general and it can be easily restated for other process algebras as well (*e.g.,* CSP [12] or π-calculus [22]).

2.1 CCS

We briefly remind some notation and concept about CCS that will be used in the following. CCS assumes a set of $\mathcal{L}$ of *actions*. Each action to represent emitted signals or (if over-lined) received ones. Actions $\overline{a}$ and a (*i.e.,* complementary send and receive actions) are called *co-actions*. The operator $^-$ is a bijection with $\overline{\overline{a}} = a$. The special symbol τ is used to model any, unobservable to the environment, *internal action*. Let $Act = \mathcal{L} \cup \overline{\mathcal{L}}$ be the set of visible actions, and $Act_\tau = Act \cup \{\tau\}$ denotes the full set of possible actions. In CCS syntax, processes are defined by the following grammar:

$$P, Q ::= \mathbf{0} \quad | \quad a.P \quad | \quad P + Q \quad | \quad P \parallel Q \quad | \quad P\backslash\mathcal{A} \quad | \quad P[f] \quad | \quad X$$

Informally $\mathbf{0}$ is the process that does not perform any action; $a.P$ is the action prefix operator with $a \in Act_\tau$; $P+Q$ can, non deterministically, choose between the behaviour the process P or that of the process Q; $\parallel$ is the parallel operator; $P\backslash\mathcal{A}$ is the action restriction operator over the set of action $\mathcal{A} \subset Act$, meaning that actions $a \in \mathcal{A}$ cannot be performed. This operator is used to specify processes which must synchronize on actions $\overline{a}$ and a. $P[f]$ is the re-naming operator, renaming each a into $f(a)$. Additionally we assume that every process identifier X has a defining equation of the form $X \stackrel{\text{def}}{=} P$ describing its behavior.

The semantics of CCS is given operationally over labeled transition systems (*LTS*) (see [20]). A *LTS* is a tuple $(\mathcal{Q}, Q_0, Act_\tau, \longrightarrow)$, whose transition relation $\longrightarrow$ defines the usual concept of derivation in one step: $P \stackrel{a}{\longrightarrow} P'$ means that process P, by executing action a, evolves in one step in the process P', while we write $P \stackrel{a}{\longrightarrow}$ to underline that P can perform action a to evolve in some process. Finally, we write $\longrightarrow_*$ the obvious transitive and reflexive closure of $\bigcup_{a \in Act_\tau} \stackrel{a}{\longrightarrow}$. In the following we let $Der(P) = \{P' | P \longrightarrow_* P'\}$, $Sort(P) = \{a : a \in Act_\tau, \exists P' \in Der(P), P' \stackrel{a}{\longrightarrow}\}$, and $\mathcal{E}_\mathcal{H} = \{X : Sort(X) \subseteq \mathcal{H}\}$, where $\mathcal{H}$ is a set of actions.

2.2 Specifying a System

Talking about formal models of fault-tolerant systems we will refer to the following notation, which is a model-oriented version of the concepts defined in [15]:

- *A System* is modelled as a finite state CCS process P, composed by a set of processes communicating each other and interacting with the environment

through the execution of actions.

- *A Failing System* is modelled as a CCS process $P_{\mathcal{F}}$ obtained by extending the processes in P with the possibility of executing particular external actions from a set $\mathcal{F}$ of *fault actions*. In other words each kind of fault is modelled explicitly as an action $f \in \mathcal{F}$. The execution of an action f corresponds to the occurrence of a fault. The specification P_f of the failing system is obtained by introducing occurrences of the possible fault action f in the model of the system P. If the action f is executed in a state of a system, then the failure mode of the system is exhibited, otherwise, the system goes on with its behaviour. For instance, $P_f = P + f.P'$ denotes the process that can behave as P' in the case of occurrence of fault f.
- *A Fault Tolerant (Candidate) System* is modelled as a process $P^{\#}_{\mathcal{F}}$, obtained by extending the failing model $P_{\mathcal{F}}$ with additional processes realizing error-recover strategies, in accordance to some fault tolerant design strategy (*e.g.,* replication of component, triple modular redundancy, voting etc.). Generally speaking, the application of tolerance techniques in modelling leads to a process $P^{\#}_{\mathcal{F}}$ that, described in the CCS notation, have the indicative form:

$$P^{\#}_{\mathcal{F}} = (P^{(1)}_{\mathcal{F}} \parallel \cdots \parallel P^{(n)}_{\mathcal{F}} \parallel Q)\backslash\mathcal{A}$$

  where (a) $P^{(i)}_{\mathcal{F}}$ is the i-th replica of $P_{\mathcal{F}}$, with eventually an appropriate renaming of actions; (b) Q is the process modelling the extras components *e.g.,* a voter; (c) $\mathcal{A} = \{a_1, \cdots, a_h\}$, $a_i \notin \mathcal{F}$, are the label of synchronisation actions between $P^{(i)}_{\mathcal{F}}$'s and Q.

Let us observe that, differently from other approaches, for example the one in [5], we will not assume any specific *fault assumption* in $P^{\#}_{\mathcal{F}}$. Precisely we did not express, in P, any assumption about the possibility of occurrence of faults, but we let fault assumptions to be characterised by the environment.

2.3 Fault Assumption Models

We now consider fault tolerant analysis as the analysis of an *open system* acting in a general and unspecified *faulty environment*. This approach has been widely applied in system security analysis (*e.g.,* see [17,18]) and we will show how it could be successfully applied in fault tolerance analysis too.

In our case the environment $F_{\mathcal{F}}$, acting as a fault injector, is able to induce any fault in the system via actions in $\mathcal{F}$. So, if we assume that the occurrence of the fault is always induced by a process $F_{\mathcal{F}}$, the schenario we are proposing is the following:

$$(P^{\#}_{\mathcal{F}} \parallel F_{\mathcal{F}})\backslash\mathcal{F} \tag{1}$$

where $P^{\#}_{\mathcal{F}}$ is a model of a candidate fault tolerant system. Again we underline that the way we model the faulty behaviour differs from [5] where, in turn, the fault-injector could also exhibit part of the system behaviour. This

clear separation between the process and the fault-injector makes the usage of partial model checking (see Section 3) feasible. Before going on it is worth to observe that in (1) fault actions are hidden (*i.e.,* restricted). This implies that $P_{\mathcal{F}}$ and $F_{\mathcal{F}}$ could synchronize only on fault events and that faults are indeed internal (*i.e.,* not observable) actions of the failing systems. This means that our analysis lays at the abstraction level where what it is really observable, in a failing system, is only its behaviour. In practice, a failing systems should be either resilient to faults or the presence of faults should be highlighted by its subsequent behaviour.

In such a framework, then the set of all the possible fault-injector processes represents our set of all possible fault assumptions:

Definition 2.1 [Fault Assumption Models] The set $\mathcal{E}_{\mathcal{F}}$ of *fault assumption models* is given by definition:

$$\mathcal{E}_{\mathcal{F}} = \{F : Sort(F) \subseteq \mathcal{F}\}$$

In other words, $\mathcal{E}_{\mathcal{F}}$ is the set of all CCS processes whose alphabet of actions is in $\mathcal{F}$.

Generally speaking a *fault tolerance* system is expected to behave correctly despite faults. Anyway different meaning of fault tolerant behaviour may be defined. For example a system may be required to be: *fail safe*, when failures cause transition in a state in which no catastrophic event can occur; *fail stop* when failures cause a stop in delivering a service, or *fail silence* when failures provide only a temporarily interruption of the services.

2.4 Equational $\mu-$calculus

Equational μ-calculus is a process logic which extends *HML* [10] with fix-point equations in order to reason directly about recursive definitions of properties. It permits us to analyse non terminating behaviour of systems. It is a powerful temporal logic which subsumes several other state based logic such as CTL, CTL^* and $ECTL^*$ [7], as well as action based logic as $ACTL$, $ACTL^*$ [23]. Equational μ-calculus is based on fix-point equations that substitute recursion operators. Let X be a variable ranging over a set $\mathcal{V}$ of variables, then a minimal (maximal) fix-point equation is $X =_\mu \phi$ ($X =_\nu \phi$), where ϕ is an *assertion*, that is a simple modal formula without recursion operators. The syntax of the assertions (ϕ) and of the lists of equations (φ) is given by the following grammar:

$$\begin{array}{ll} \textit{assertion} & \phi := X \mid \mathtt{tt} \mid \mathtt{ff} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a]\phi \\ \textit{equations list} & \varphi ::= (X =_\nu \phi)\ \varphi \ \mid (X =_\mu \phi)\ \varphi \ \mid \epsilon \end{array}$$

It is assumed that variables appear only once on the left-hand sides of the equations of the list, the set of these variables will be denoted as $Defs(\varphi)$. A

list of equations is closed if every variable that appears in the assertions of the list is in $Defs(\varphi)$.

Let be $\mathcal{M} = (\mathcal{Q}, Q_0, Act_\tau, \longrightarrow)$ a *LTS* and ρ an environment function that assigns subset of $\mathcal{Q}$ to the variables appearing in the assertion of φ out of $Defs(\varphi)$. As notation, we use in the following $\sqcup$ to represent union of disjoint environments. Let σ be in $\{\mu, \nu\}$, then $\sigma U.f(U)$ represents the σ fix-point of the function f in one variable U. The semantics $[\![\varphi]\!]_\rho$ of the equation list is an environment which assigns subsets of $\mathcal{Q}$ to variables in $Defs(\varphi)$:

$$[\![\epsilon]\!]_\rho = []$$
$$[\![(X =_\sigma \phi)\varphi]\!]_\rho = [\![\varphi]\!]_{(\rho \sqcup [U' \setminus X])} \sqcup [U'/X]$$

where $U' = \sigma U.[\![\phi]\!]'_{(\rho \sqcup [U/X] \sqcup \rho'(U))}$, and $\rho'(U) = [\![\varphi]\!]_{(\rho \sqcup [U/X])}$. The semantics $[\![\phi]\!]'_\rho$ of an assertion ϕ is the same as for the μ-calculus:

$$[\![\mathtt{tt}]\!]'_\rho = \mathcal{Q}, \quad [\![\mathtt{ff}]\!]'_\rho = \emptyset, \quad [\![X]\!]'_\rho = \rho(X), \quad [\![X]\!]'_\rho = \rho(X)$$
$$[\![\phi_1 \wedge \phi_2]\!]'_\rho = [\![\phi_1]\!]'_\rho \cap [\![\phi_2]\!]'_\rho, \quad [\![\langle a \rangle \phi]\!]'_\rho = \{Q | \exists Q' : Q \xrightarrow{a} Q' \text{ and } Q' \in [\![\phi]\!]'_\rho\}$$
$$[\![\phi_1 \vee \phi_2]\!]'_\rho = [\![\phi_1]\!]'_\rho \cup [\![\phi_2]\!]'_\rho, \quad [\![[a] \phi]\!]'_\rho = \{Q | \forall Q' : Q \xrightarrow{a} Q' \text{ implies } Q' \in [\![\phi]\!]'_\rho\}$$

Informally $[\![(X =_\sigma \phi)\varphi]\!]_\rho$ says that the solution to $(X =_\sigma \phi)\varphi$ is the σ fix-point solution U' of $[\![\phi]\!]'_\rho$ where the solution to the rest of the list of equations φ is used as environment. We write $Q, \mathcal{M} \models \varphi \downarrow X$ as notation for $Q \in [\![\varphi]\!](X)$ when the environment ρ is evident from the context or φ is a closed list (*i.e.*, without free variables); furthermore X must be the first variable in the list φ.

3 Logic Characterisation of Fault Tolerance

This section explains how the fault tolerance verification problem can be characterised as a validation problem of logic formulas. This characterisation is prevalently based on a technique called *partial model checking* [3], while the automatic verification is possible afterwards by theorem proving. It is worth noticing that the same approach has been proposed for security analysis [17,18].

3.1 Partial Model Checking

This technique relies upon compositional methods for proving properties of concurrent processes specified in terms of a *process algebra*. It has been presented in this formulation by Andersen [3]. In the following we use $P \|_\mathcal{A} Q$ as a short cut for $(P \parallel Q) \setminus \mathcal{A}$. The intuitive idea underlying the partial evaluation is the following: proving that $P \|_\mathcal{A} Q$ satisfies φ is equivalent to prove that Q

satisfies a modified formula $\varphi /\!\!/_{\mathcal{A}} P$, where $/\!\!/_{\mathcal{A}} P$ is the partial evaluation function for the parallel composition operator (see [3]). In Fig. 1 we will describe the partial evaluation function for the CCS operator $\|_{\mathcal{A}}$ where $\mathcal{A} \subseteq Act$.

In [1,3,16], we can also find the proofs of the following lemma:

Lemma 3.1 *Given a process $P\|_{\mathcal{A}} Q$ (where P is finite-state) and an equational specification $\varphi \downarrow X$ we have:*

$$P\|_{\mathcal{A}} Q \models (\varphi \downarrow X) \text{ iff } Q \models (\varphi \downarrow X) /\!\!/_{\mathcal{A}} P$$

The size of the formula obtained after the partial model checking procedure is polynomial in the size of the process and the formula. Remarkably, this function is exploited in [3] to perform model checking efficiently *i.e.,* both P and Q are specified. In our setting, the process Q will be not specified.

Supposing $\mathcal{M} = (\mathcal{Q}, Q_0, Act_\tau, \rightarrow\}$ be a finite state *LTS*, where $\mathcal{Q} = \{Q_0, \ldots, Q_n\}$:

$$\begin{array}{lcl}
(\varphi \downarrow X) /\!\!/_{\mathcal{A}} \mathcal{M} & = & (\varphi /\!\!/_{\mathcal{A}} \mathcal{M}) \downarrow Q_0, \\
\epsilon /\!\!/_{\mathcal{A}} \mathcal{M} & = & \epsilon \\
(X =_\sigma \phi)\varphi /\!\!/_{\mathcal{A}} \mathcal{M} & = & ((X_Q =_\sigma \phi /\!\!/_{\mathcal{A}} Q)_{Q \in \{Q_0,\ldots,Q_n\}})(\varphi) /\!\!/_{\mathcal{A}} \mathcal{M} \\
X /\!\!/_{\mathcal{A}} Q & = & X_Q \\
\langle a \rangle \phi /\!\!/_{\mathcal{A}} Q & = & \langle a \rangle (\phi /\!\!/_{\mathcal{A}} Q) \vee \bigvee_{Q \xrightarrow{a} Q'} \phi /\!\!/_{\mathcal{A}} Q', \quad \text{if } a \neq \tau \wedge a \notin \mathcal{A} \\
\langle a \rangle \phi /\!\!/_{\mathcal{A}} Q & = & \mathtt{ff}, \quad \text{if } a \in \mathcal{A} \\
\langle \tau \rangle \phi /\!\!/_{\mathcal{A}} Q & = & \langle \tau \rangle (\phi /\!\!/_{\mathcal{A}} Q) \vee \bigvee_{Q \xrightarrow{\tau} Q'} \phi /\!\!/_{\mathcal{A}} Q' \vee \bigvee_{Q \xrightarrow{a} Q'} \langle \overline{a} \rangle (\phi /\!\!/_{\mathcal{A}} Q') \\
\mathtt{ff} /\!\!/_{\mathcal{A}} Q & = & \mathtt{ff}
\end{array}$$

Fig. 1. The partial evaluation function for $\|_{\mathcal{A}}$. We have left out the cases for $[a]\phi$, $\wedge$ and $\mathtt{tt}$. They are immediate duals of $\langle a \rangle \phi$, $\vee$ and $\mathtt{ff}$.

3.2 Fault Tolerance Analysis Through Partial Model Checking

This section shows how to formalize fault tolerance using partial model checking. We start considering a system model P and its fault-tolerant version $P^{\#}_{\mathcal{F}}$. Let us suppose φ be an equational μ-calculus formula expressing some desirable behaviour of a system even in presence of faults. In a fault tolerance analysis we are interested in understanding in which *fault assumption* $F_{\mathcal{F}}$, the specification $P^{\#}_{\mathcal{F}}$ preserves φ. The set of such fault assumption can be formalized as:

$$\mathfrak{F}^P_\varphi = \{F_{\mathcal{F}} : (P^{\#}_{\mathcal{F}} \parallel F_{\mathcal{F}}) \backslash \mathcal{F} \models \varphi\} \tag{2}$$

The previous set may be considered a characterization of the Fault Tolerance of $P^{\#}_{\mathcal{F}}$. Indeed [4], if $\mathcal{E}_{\mathcal{F}} = \mathfrak{F}^{P}_{\varphi}$ then it means that no fault assumption is able to force $P^{\#}_{\mathcal{F}}$ not to satisfy φ.

Definition 3.2 [Logic Characterisation of Fault Tolerance 1] A process $P^{\#}_{\mathcal{F}}$ is *fault tolerant* w.r.t. *the logical property* φ if and only if $\mathcal{E}_{\mathcal{F}} = \mathfrak{F}^{P}_{\varphi}$.

However, due to its implicit definition, Def. (2) has not a practical usage. Here, we exploit the partial model checking techniques to find a more suitable characterization for (2). In fact, we can write:

$$\mathfrak{F}^{P}_{\varphi} = \{F_{\mathcal{F}} : F_{\mathcal{F}} \models \varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}\} \tag{3}$$

where $\varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$ is obtained by using partial model checking of φ *w.r.t.* $P^{\#}_{\mathcal{F}}$.

Such a definition of $\mathfrak{F}^{P}_{\varphi}$ is easier to be managed since corresponds to common representation of sets and permits to define as a validity checking problem the fault tolerance of a process *w.r.t.* a formula. At this point we can formulate our logic characterization of fault tolerance.

Definition 3.3 [Logic Characterisation of Fault Tolerance 2] We say that a process $P^{\#}_{\mathcal{F}}$ is *fault tolerant* w.r.t. *the logical property* φ if and only if $\varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$ is a valid formula *w.r.t.* processes in $\mathcal{E}_{\mathcal{F}}$.

It is easy to prove that:

Proposition 3.4 *A process $P^{\#}_{\mathcal{F}}$ enjoys Definition 3.2 if and only if it enjoys Definition 3.3.*

4 Analysis techniques

We have shown how the *fault tolerance* of $P^{\#}_{\mathcal{F}}$ *w.r.t.* the property φ may be expressed as a validity statement in the μ-calculus. Thus one could use either the standard validity checking algorithms (*e.g.*, see [25]) or the proof system developed in [27]. Clearly, the complexity of checking *fault tolerance* of a system $P^{\#}_{\mathcal{F}}$ *w.r.t.* a formula φ in any fault assumption turns out to be exponential in the size of the formula obtained after the partial evaluation, which is in turn polynomial in the size of the system $P^{\#}_{\mathcal{F}}$ and φ.

4.1 Fault Tolerance Analysis as Theorem Proving

This complexity cost has however the advantage of making very simple the analysis of a system $P^{\#}_{\mathcal{F}}$ *w.r.t.* a wide class of fault assumption scenarios. In particular, such scenarios could be denoted by another μ-calculus formula $\varphi_{\mathcal{F}}$. The verification problem is still a validity checking one with respect to the formula $\varphi_{\mathcal{F}} \Rightarrow \varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$. Formally:

[4] Here $\mathcal{E}_{\mathcal{F}}$ is the set of all fault assumption, see Definition 2.1.

Definition 4.1 [Logic Characterisation of Fault Tolerance 3] Let $\varphi_{\mathcal{F}}$ to be the characteristic formula for a set of fault scenarios. We say that a process $P^{\#}_{\mathcal{F}}$ is *fault tolerant* w.r.t. *the logical property* φ if and only if $\varphi_{\mathcal{F}} \Rightarrow \varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$ is valid.

4.2 Fault Tolerance against a fixed Scenario

We recall that partial model checking has been advocated as an efficient method for performing model checking by Andersen. Assume to have a given fault scenario corresponding to a finite-state CCS process. Now let us assume that $P^{\#}_{\mathcal{F}}$ is indeed a parallel composition of k processes $P_1 \| \ldots \| P_k \| \mathbf{0}$, and testing if it satisfies a formula φ' is easy by applying several times the partial model checking function.

In fact we obtain a formula φ'_1 that $P_2 \| \ldots \| P_k \| \mathbf{0}$ must satisfy. Here, one could apply some logical simplifications on the formula φ'_1, in order to obtain a smaller and so more tractable formula. After k applications of the partial model checking, we obtain that the stuck process $\mathbf{0}$ must satisfy a formula φ'_k. The model checking of a formula *w.r.t.* the stuck process is usually very efficient. Thus, the model checking of a system which consists of k parallel-running processes is reduced, through partial model checking, to the model checking of the stuck process $\mathbf{0}$.

4.3 Universal formulas

A way to obtain a more efficient analysis is to consider a variant of *universal* μ-calculus formulas (*e.g.,* see [11]). Here, we specify this class for the equational μ-calculus. When a *universal* equational is translated into the standard one through the application of the Bekič theorem in [4], we get standard *universal* formula. A nice feature of this class of formulas is that their validity problem is co-NP-complete ([11]) rather than EXP-time ([25]). Moreover, a subclass of such formulas *i.e.,* the conjunctive ones, has a validity problem whose complexity is linear in the size of the formula.

Definition 4.2 The set of *universal formulas* ($\forall MC$, for short) is a class of the equational μ-calculus formulas where no $\langle a \rangle$ operator is present. The set of *universal conjunctive formulas* ($\forall_c$MC, for short) is the subset of $\forall MC$ formulas containing only conjunctions.

Several interesting properties may be written as *universal formulas i.e.,* the safety ones. The key property is that partial model checking of *universal* formulas *w.r.t.* contexts as the ones used in (3) gives again *universal* formulas.

Proposition 4.3 *Given a* universal formula ϕ*, a fault tolerant process* $P^{\#}_{\mathcal{F}}$*, a fault injector* $F_{\mathcal{F}}$ *then*

$$(P^{\#}_{\mathcal{F}} \| F_{\mathcal{F}}) \backslash \mathcal{F} \models \varphi \quad \textit{if and only if} \quad F_{\mathcal{F}} \models \varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$$

and $\varphi /\!\!/_{\mathcal{F}} P^{\#}_{\mathcal{F}}$ *is a* universal formula.

The proof relies on the fact that the partial model checking of the $[a]\phi$ operator gives a formula which consists of a conjunction of such operators. The same proposition holds for the *universal conjunctive* ones. The nice feature of these formulas is that their validity problem is the complementary one of the satisfiability problem of the negated of such formulas that turns out to be a disjunctive one in the sense of [13], whose satisfiability problem can be decided in linear time in the size of the formula.

4.4 Linear time formulas

As a final consideration we can observe that, when the formula under examination φ is a linear time formula, i.e. that holds for a system iff it holds in each trace of the system, we can reduce the fault tolerance problem *w.r.t.* such a formula to a fault tolerance problem *w.r.t.* a specific fault assumption, precisely:

$$Top_{\mathcal{F}} = \sum_{f \in \mathcal{F}} f.Top_{\mathcal{F}} + \overline{f}.Top_{\mathcal{F}}$$

Consider the set Φ of linear time formulas that are true in a model if and only if are true in each computation path take as a model. Several interesting fault-tolerance properties are in this class.

Proposition 4.4 $P^{\#}_{\mathcal{F}}$ *is* fault tolerant w.r.t. $\varphi \in \Phi$ *if and only if* $P^{\#}_{\mathcal{F}}$ *is* fault tolerant w.r.t. φ *against only* $Top_{\mathcal{F}}$.

5 Example

As an example, we will illustrate a CCS model of a Triple Modular Redundancy (TMR) system, taken from [2]. For sake of conciseness we will write our example in a value passing style CCS, which is knows to be a shortcut for CCS. The basic component is a simple module implementing the identity function, which naively returns the same value received in input. Here we assume that values are taken from a set of binary values *Val*. The actions `mi` and `mo` are used to communicate module inputs and outputs.

$$P \stackrel{\text{def}}{=} mi(x).\overline{mo}(x).P$$

In modelling the corresponding failing module we assume that in case of fault any value from *Val*, it may be produced in output. In accordance to our modelling framework, the fault is assumed to be caused by a special fault action f triggered by the environment.

$$P_f \stackrel{\text{def}}{=} mi(x).(\overline{mo}(x) + f.\sum_{y \in Val} \overline{mo}(y)).P_f + f.(mi(x).\sum_{y \in Val} \overline{mo}(y)).P_f$$

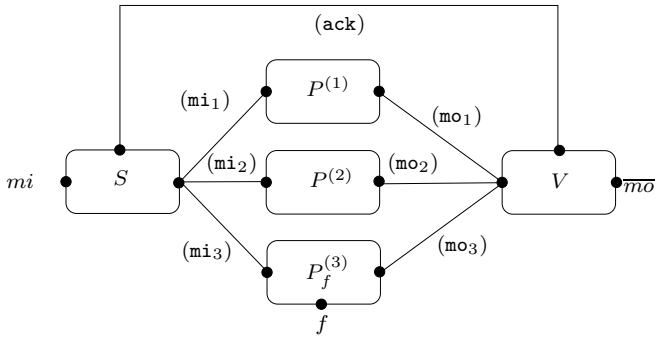

Fig. 2. The flow diagram of the TMR version of P

Note that in modelling the failing behaviour we have assumed that a fault may happen everywhere in the module. Different faulty behaviours can be taken into consideration, for example by assuming a module to fail only after inputs. The (candidate) fault system $P_f{}^{\#}_{\mathcal{F}}$, may be designed using the classic solution of Triple Modular Redundancy, where three instances of the module P_f are composed with the additional modules of a splitter S and a voter V. We want the splitter to deliver the input value to each of the three instances of the module P_f, respectively $P_f^{(1)}, P_f^{(2)}$ and $P_f^{(3)}$, so we define an indexed version of the module:

$$P_f^{(i)} \stackrel{\text{def}}{=} P_f[\mathtt{mi}_i/\mathtt{mi}, \mathtt{mo}_i/\mathtt{mo}]$$

An indexed from P can be defined as for P_f. If we chose the splitter S to deliver the input value to the three modules in a specified order, we have

$$S \stackrel{\text{def}}{=} mi(x).\overline{mi}_1(x).\overline{mi}_2(x).\overline{mi}_3(x).ack.S$$

Note that a splitter delivering the input value in an arbitrary order may be easily modelled. The voter:

$$V \stackrel{\text{def}}{=} mo(x_1).mo(x_2).mo(x_3).(\textbf{if } x_1 = x_2 \textbf{ then } \overline{mo}(x_1) \textbf{ else } \overline{mo}(x_3)\ ).\overline{ack}.V$$

The resulting fault tolerant version $P_f^{\#}$ of P, is its TMR version where, for example, a single faulty module is inserted (see also Figure 2).

$$P_f{}^{\#} \stackrel{\text{def}}{=} (S \parallel P^{(1)} \parallel P^{(2)} \parallel P_f^{(3)} \parallel V) \setminus \{\mathtt{mi}_1, \mathtt{mi}_2, \mathtt{mi}_3, \mathtt{mo}_1, \mathtt{mo}_2, \mathtt{mo}_3, \mathtt{ack}\}$$

The strategies illustrated in this paper, or a combination of them, can be applied on $P_f{}^{\#}$ to verify if it satisfy a property expresses as an equational μ-calculus formula φ. The choise of the most suitable verification strategy usually depends on the structure of the φ and on the fault scenario against whom to check it, as explained in Section 4.

6 Conclusions

This paper first shows how a fault tolerant (candidate) system may be formalized using a process algebras (CCS). The formal model is built following a uniform modeling scheme requiring both the failing behaviour (with respect to fault occurrences) and fault-recovering procedures to be specified. Faults are modeled as specific actions in the system model, that a fault injector process is able to activate. Fault tolerant properties are expressed as equational μ-calculus formulae.

This general framework makes a logical characterization of fault tolerance possible. In fact the fault tolerant verification problem (*w.r.t.* a property) can be stated as a module checking problem, that is as the verification of an open system in an environment able to inject any fault.

Moreover exploiting partial model checking strategies, the fault tolerant verification problem (*w.r.t.* a formula) may be expressed as a validity problem of a new formula obtained by partial evaluation of the system model. In this way, general validation tools and proof techniques can be exploited. For a more efficient (and tailored) analysis we have proposed, for example, the use of universal and conjunctive μ-calculus formulas whose validity problem is easier to be solved. As a future work, we are developing a semi-automated tool which, based on an existing partial model checker (*e.g.,* see [16]), would be able to perform the fault tolerant analysis so far described.

References

[1] H. R. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, 1993.

[2] G. Bruns. *Distributed System Analysis with CCS*. Prentice Hall, 1997.

[3] H. R. Andersen. Partial model checking (extended abstract). In *Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[4] H. Bekič. Definable operations in general algebras, and the theory of automata and flow charts. In C.B. Jones, editor, *Hans Bekič: Programming Languages and Their Definition*, volume 177 of *LNCS*, pages 30–55. Springer-Verlag, 1984.

[5] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability (STVR)*, 12(4):251–275, December 2002.

[6] J. Bradfield and C. Stirling. *Modal Logics and μ-calculi: an introduction*, pages 293–332. Handbook of Process Algebra. Elsevier, North-Holland, 2001.

[7] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[8] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[9] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 3–12. IEEE Computer Society, 2002.

[10] M. Hennessy and R. Milner. Algebraic laws for non determinism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[11] T. A. Henzinger, O. Kupferman, R. Majumdar On the Universal and Existential Fragments of the mu-Calculus In Proc. of TACAS, pages 49-64, LNCS 2619, 2003.

[12] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[13] D. Janin and I. Walukiewicz. Automata for the μ-calculus and related results. In J. Wiedermann and P. Hájek, editors, *Proceedings 20th Intl. Symp. on Mathematical Foundations of Computer Science, MFCS'95*, volume 969 of *LNCS*, 1995.

[14] E. Jonsson. An integrated framework for security and dependability. In *Proceedings of the New Security Paradigms Workshop*, pages 22–25, Charlottesville, VA, USA, September 1998.

[15] J. C. Laprie. Dependability: Basic concepts and terminology. In J. C. Laprie, editor, *Dependable Computing and Fault-Tolerant Systems*, volume 5. Springer-Verlag, 1995.

[16] J. Lind-Nielsen. Mudiv: A program performing partial model checking. Master's thesis, Department of Information Technology, Technical University of Denmark, September 1996.

[17] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *Proc. of the 11th IEEE Computer Security Foundation Workshop*, pages 44–52. IEEE, Computer Society, 1998.

[18] F. Martinelli. Analysis of security protocols as open systems. TCS 290(1): 1057-1106 (2003)

[19] C. Meadows and J. McLean. Security and dependability: Then and now. In *Proc. of Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solutions - Workshop II*, Williamsburg, VA, November 1998.

[20] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[21] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, N.Y., 1990.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[23] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *Lecture Notes in Computer Science*, volume 469, pages 407–419. Springer-Verlag, 1990.

[24] A. Simpson, J. Woodcock, and J. Davis. Safety through security. In IEEE Computer Society Press, editor, *Proc. of the 9th International Workshop on Software Specification and Design*, pages 18–24, Ise-Shima. Japan, April 1999.

[25] R. S. Streett and E. A. Emerson. An automata theoretic procedure for the propositional $\mu-$calculus. *Information and Computation*, 81(3):249–264, 1989.

[26] H. H. Thompson, J. A. Whittaker, and F. E. Mottay. Software security vulnerability testing in hostile environments. In ACM Press, editor, *Proceedings of the 2002 ACM symposium on Applied computing (SAC 02*, pages 260–264, Madrid, Spain, 2002.

[27] I. Walukiewicz. Completeness of Kozen's axiomatization of the propositional λ-calculus. In *Symposium on Logic in Computer Science (LICS '95)*, pages 14–24, Los Alamitos, Ca., USA, 1995. IEEE Computer Society Press.