

Model Checking Fault Tolerant Systems

Cinzia Bernardeschi¹, Alessandro Fantechi², Stefania Gnesi³

¹ Dipartimento di Ingegneria della Informazione, Univ. di Pisa
Via Diotisalvi 2, 56126, Pisa, Italy, phone:+39-50-568511 *cinzia@iet.unipi.it*

² Dip. di Sistemi e Informatica, Univ. di Firenze
Via S. Marta 3, 50139, Firenze, Italy, phone:+39-55-4796265 *fantechi@dsi.unifi.it*

³ Istituto di Elaborazione dell'Informazione, IEI-CNR
Via G. Moruzzi 1, 56124 Pisa, Italy, phone:+39-50-3152918 *gnesi@iei.pi.cnr.it*

Keywords: formal methods, fault tolerance, model checking, verification.

Abstract

This paper proposes a modelling approach suitable for formalising fault tolerant systems, taking into account different fault scenarios. Verification of properties of such systems is then performed using model checking. A general framework for the formal specification and verification of fault tolerant systems is defined starting from these principles, and an experience in its application to two case studies is then presented.

1 Introduction

The large deployment of computer-controlled systems has raised in last years many concerns about safety issues when human activities and lives depend on them. A combination of fault prevention, fault tolerance, fault removal and fault forecasting techniques are commonly used in order to achieve a high degree of dependability [33]. There not exists, however, a common agreement on a standard method to combine and integrate individual techniques: industries, basing on their different backgrounds and application fields, adopt their own, different, development trajectories, which are based on the separate use of various techniques aimed at enhancing dependability. Indeed, the combination and integration of different dependability techniques is still an open research area.

This paper addresses the combination of the adoption of fault tolerance mechanisms and of the use of formal methods, and in particular formal verification tools, in the development of a system. While fault tolerance is achieved through a set of well-established and commonly adopted techniques, which often exploit hardware redundancy, formal methods have not gained a wide acceptance as a viable means to reduce the failure rate of programs, though several success stories have been reported (see, for example, [20]), and international standards and guidelines (e.g. the CENELEC EN50128 guidelines for software development in the railway industry [13]) recommend the use of formal methods in the development of safety critical computer-controlled systems.

Nowadays, the industrial trend is directed to the adoption of formal verification techniques to validate the design, integrating them within the existing development process. Industries are more keen to accept formal verification techniques assessing

the quality attributes of their products, obtained by a traditional life cycle, rather than a fully formal life cycle development, due to the lower training and innovation costs of the former.

Following this trend, the paper proposes the use of a formal verification technique, namely model checking, to verify the conformance of a design with respect to desired properties, such as:

1. *Correctness*. The system delivers a correct service (in absence of faults)
2. *Fault tolerance*. The system delivers a correct service, despite faults
3. *Fail-silence*. The system failures can only be omission failures, that is, failures to temporarily provide the service to the user of the system
4. *Fail-stop*. In case of faults, the system terminates the delivery of its service
5. *Fail-safe*. The system failure is a transition to a state in which no catastrophic event can occur

The properties 2,3,4 and 5 will be studied with respect to specific classes of faults and in presence of given fault occurrences, that is, under well-defined *fault assumptions*. The properties informally expressed above can be formally specified using some logic formalism; temporal logic, whose operators permit explicit quantification over all possible futures, is a possible candidate. If a formal model of the system under analysis is generated, typically by means of state machines or transition systems, model checking algorithms can be used to prove that the model of the system satisfies the properties expressed in a temporal logic [14]. Unfortunately, when model checking is applied to a system composed of several subsystems, it suffers of the so called "State Space Explosion" problem. In this case a finite state model with a number of states which is exponential in the number of the component subsystems can be generated. The redundancy often introduced by fault tolerance mechanisms could be a possible cause of such a problem, since it increases (often even duplicates or triplicates) the number of subsystems. In this paper it is shown that instead some typical redundant structures can help to contain the increase in the state space. Following these observations, suitable techniques to adopt in order to address this problem are indicated.

The paper is organised as follows. Section 2 presents a technique adopted for formally specifying fault tolerant systems. Section 3 addresses fault tolerant systems properties, their formalisation and verification. Section 4 deals with the characteristics that make model checking technique applicable in this field. Section 5 reports on the application of the proposed formalisation technique and verification tools to two case studies concerning real systems. Section 6 reviews related work.

2 Modelling Fault tolerant Systems

This section presents an approach to specify fault tolerant systems in such a way that the specification can be analysed by model checking techniques. The approach is derived from [4], in which the following concepts were defined, in accordance with the terminology proposed in [33]:

Definition 1 (system) *System denotes the specification of the system in absence of faults.*

Definition 2 (failure mode) *Failure mode denotes the way the system fails, in terms of the behaviour of the system after the occurrence of a fault.*

Definition 3 (failing system) Failing system *denotes the complete specification of the system, including all possible occurrence of faults and the corresponding failure modes.*

Definition 4 (fault tolerant system) Fault tolerant system *denotes the specification of the addition of some fault tolerance technique to a failing system.*

Definition 5 (fault assumption) Fault assumption *denotes the assumptions made on the effectively possible occurrence of faults in the system.*

The approach presented is based on the following points:

- a system is modelled as set of processes which communicate each other and interact with the environment by executing actions
- faults are modelled directly by actions of the processes themselves. For each fault action, the relative failure mode is also specified. For example, a crash fault in a state extends the behaviour of the system by allowing a crash to occur in that state. Moreover, faults are modeled as random events
- assumptions on the occurrence of faults are included in the specification by defining ad hoc fault assumption processes. This allows the behaviour of the fault tolerant system to be studied under different fault scenarios

2.1 Specifying a system

Two different formalisms are interchangeably used to specify a system: the CCS/Meije process algebra and an (almost) equivalent graphical notation. The choice of these formalisms, mainly due to the availability of verification tools, has proven valuable for their ability of modeling fault assumptions and fault tolerance mechanisms.

CCS/Meije is actually the subset of the Meije process algebra [1], in which only the parallel composition operator that corresponds to the CCS one [34] is considered.

The syntax of CCS/Meije permits a two-layered specification of concurrent systems, as process terms. The first layer is related to sequential processes, the second one to networks of parallel sub-processes, supporting communication and action renaming or restriction.

The CCS/Meije syntax uses a set of labels Act as atomic actions names ranged over by α, β, \dots ; such names represent emitted signals if they are prefixed by the "!" character, or received ones if they are prefixed by "?". Actions $!\alpha$ and $?\alpha$ are called co-actions. τ denotes a special action not belonging to Act , the unobservable action used to model internal process actions. $Act_\tau = Act \cup \{\tau\}$, ranged over by a, b, \dots , denotes the full set of actions that a process can perform.

The syntax of the language is the following:

$$R ::= \mathbf{stop} \mid X \mid a : R \mid R + R \mid \mathbf{let\ rec} \{X = R \mid \mathbf{and} \ X = R\} \mathbf{in} \ X$$

$$P ::= R \mid P \parallel P \mid P \setminus \alpha \mid P[\alpha/\beta] \mid \mathbf{let} \{X = P \mid \mathbf{and} \ X = R\} \mathbf{in} \ X$$

where

- R is the syntactic category of sequential processes and P is the syntactic category of networks of parallel processes
- $[\dots]$ denotes an optional and repeatable part of the syntax

- **stop** is the process that does not perform any action
- $a : R$ is the action prefix operator: the action a is executed and then the process behaves like R
- $X = R$ bounds the process variable X to the process R
- the sum is the non deterministic choice operator: a process $R1 + R2$ can choose between the behaviour the process $R1$ and that of the process $R2$
- the **let rec** construct allows recursive definitions of process variables
- \parallel is the parallel operator. This operator is used to specify the interleaved execution of processes and their possible synchronisation when co-actions are executed.
- $P \setminus \alpha$ is the action restriction operator, meaning that α can only be performed within a communication. This operator is used to specify processes which must synchronise on actions $!\alpha$ and $?\alpha$. The restriction operator transforms the couple of co-actions executed together into the internal action τ
- $P[\alpha/\beta]$ is the substitution operator, renaming β into α

The semantics of CCS/Meije is given operationally over LTSs. An LTS consists of a set of states and transitions between states, where a transition corresponds to the execution of an action of the system.

Definition 6 *An LTS is a 4-tuple $\mathcal{A} = (Q, q^0, Act_\tau, \rightarrow)$, where: Q is a set of states; q^0 is the initial state; Act is a finite set of observable actions; $\rightarrow \subseteq Q \times Act_\tau \times Q$ is the transition relation; an element $(r, a, q) \in \rightarrow$ is called a transition and is written as $r \xrightarrow{a} q$. It denotes the transition from the state r to the state q by executing action a .*

Paths over the LTS \mathcal{A} are introduced. A sequence $\pi = (q_0, a_0, q_1) (q_1, a_1, q_2) \dots$ with $(q_i, a_i, q_{i+1}) \in \rightarrow$ is called a path from q_0 . The empty path consists of a single state $q \in Q$ and is denoted by q . A path that cannot be extended (i.e., is infinite or ends in a state without outgoing transitions) is called a *full path*. The starting state q_0 of the sequence is denoted by $first(\pi)$ and the last state of the sequence, if the sequence is finite, is denoted by $last(\pi)$. If π is an empty path (i.e. $\pi = q$), $first(\pi) = last(\pi) = q$. Concatenation of paths is denoted by juxtaposition: $\pi = \rho\theta$; it is only defined if ρ is a finite path and $last(\rho) = first(\theta)$. Let $\pi = \rho\theta$. In this case θ is a *suffix* of π and θ is a *proper suffix* if $\rho \neq q$.

Figure 1 shows the structural operational semantics of some CCS/Meije operators previously described, in terms of LTSs ¹. In particular, only finite state LTSs are considered here, since the two layered syntax of CCS/Meije adopted above allows only finite state processes to be defined².

As an example, consider the specification of a simple system that controls the position of a level crossing gate **p**, allowing an operator to start the procedures for the opening and the closure of the gate. The system is composed of three processes, the process **gate_contr_p** (the gate), the process **open_p** (the opening procedure) and the process **close_p** (the closure procedure). The process **gate_contr_p** has

¹CCS/Meije inherits the operational rules of the parallel operator from CCS, whereas the Meije parallel operator, instead, has an additional rule allowing product of actions that are not necessarily co-actions.

²The restriction to finite state systems in our opinion does not limit the applicability of the approach to fault tolerant control systems, since they are usually required to exhibit a finite-state behaviour even in presence of faults

Operator	Operational rules
$a : P$	$\frac{}{a : P \xrightarrow{a} P}$
$P + Q$	$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \quad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$
$P \parallel Q$	$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \quad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} \quad \frac{P \xrightarrow{?a} P', Q \xrightarrow{!a} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$

Figure 1: Operational semantics of some CCS/Meije operators

three states: undefined (initial state), open and closed. The gate changes its state on receiving a command from the other processes. The `open_p` operation checks the state of the gate. If the state is undefined or closed, it sends the set state on signal (action `!s_on_p`) to the gate. Similarly, the `close_p` operation checks the state of the gate. If the state is undefined or open, it sends the set state off signal (action `!s_off_p`) to the gate. The gate process is able to execute the actions `!on_p`, `!off_p` and `!undefined_p` to indicate that its current state is open, closed or undefined, respectively.

Figure 2 reports the CCS/Meije specification of `gate_contr_p`. Its states are called `UNDEFINED_P`, `ON_P` and `OFF_P`. For example, when the gate is in the state `ON_P`, the gate can send the signal `!on_p` indicating the current state of the process, receive a signal `?s_off_p` (set state off) and changing its state, or receive a signal `?s_on_p` (set state on) and remaining in the same state.

```

gate_contr_p =
let rec {
  ON_P = !on_p : ON_P +
        ?s_on_p : ON_P +
        ?s_off_p : OFF_P
and
  OFF_P = !off_p : OFF_P +
        ?s_off_p : OFF_P +
        ?s_on_p : ON_P
and
  UNDEFINED_P = !undefined_p : UNDEFINED_P +
        ?s_off_p : OFF_P +
        ?s_on_p : ON_P
} in UNDEFINED_P;

```

Figure 2: The `gate_contr_p` specification

For shortness, the specifications of the `open_op` and `close_op` processes (which would require more information on how the external environment commands the operations of the level crossing gate) are omitted here. The specification of the whole system (`net`) is given by the parallel composition of the three processes (see Figure 3). The `open_op` and `close_op` processes are independent from each other, but both must synchronise with the process `gate_contr_p` when checking the level crossing position (actions: `on_p`, `off_p`, `undefined_p`) or when commanding the change of the level crossing state (actions: `s_off_p` and `s_on_p`).

The graphical notation defined for the ATG tool [39], can alternatively be used.

```

net =
((open_op||close_op)||
  gate_contr_P\s_on_p\s_off_p\on_p\off_p\undefined_p;

```

Figure 3: The `net` specification

This notation expresses a sequential process by drawing the LTS representing its behaviour and expresses communicating processes by drawing a network of LTSs. In the first case, circles and edges are used to represent states and transitions, respectively. The initial state of the LTS is represented by a double circle and labels can be associated both to edges and to vertices. Communicating processes are represented by boxes with ports at the border. The ports are the places of inter-connection of processes with the environment. If two boxes are drawn at the same level, they can synchronise via the complementary actions they execute by linking the corresponding ports.

Figures 4 and 5 report the graphical specification of the `gate_contr_p` process and of the network corresponding to the specification shown in Figure 3, respectively. Note that the synchronisation on the action `s_on_p` between the processes `open_op` and `gate_contr_p` is modeled by linking the `!s_on_p` of the `open_p` labeled box to the port `?s_on_p` of the `gate_contr_p` labeled box.

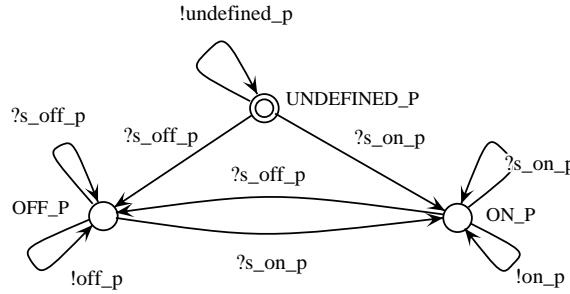


Figure 4: The `gate_contr_p` graphical specification.

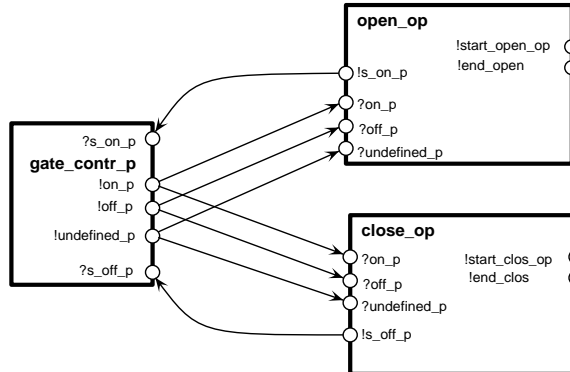


Figure 5: A network of processes

The graphical formalism provides two additional features with respect to CCS/Meije:

- Observable synchronisation actions. According to the CCS/Meije parallel operator, synchronisations become the invisible τ action. To observe synchronisation actions, a label must be put on the edge linking the ports. In this way each time a synchronisation occurs, a transition with the name of the label is shown. An example is shown in Figure 6: by setting the label L on the edge linking ports $!b$ and $?b$, each time processes synchronise by executing $!b$ and $?b$, L is observed.
- Synchronisation among three or more subsystems. This is carried out by the "web" operator. The ports corresponding to the actions which must be executed all together are linked to the web by edges. As an example, Figure 6 shows a multi-way synchronisation among processes P , Q and R . A web is used in Figure 6 to synchronise the three subsystems on the action f .

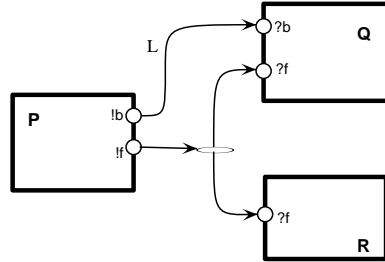


Figure 6: Two-way and multi-way synchronisation

Given a network of LTSs or a process algebra term, the generation of the LTS representing its overall behaviour is automatically performed by means of tools, based on the related operational semantics rules [6].

2.2 Specifying the failing system

Each kind of fault is modelled explicitly as an action. The execution of the action corresponds to the occurrence of the fault. Let \mathcal{F} be the set of actions modelling the possible faults in a system. The specification of the failure of the system is obtained by introducing occurrences of the possible faults as transitions in the LTSs modeling the system. If the action $f \in \mathcal{F}$ 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.

Figure 7 models the failing system `gate_contr.p`, when two kind of faults are considered: a permanent fault, modelled by the `?f_p` action, and a temporary fault, modelled by the `?f_t` action.

The permanent fault leads the system to a special state named `FAULTY_P` in which the state of the system is undefined forever (action `!undefined_p`). The temporary fault causes the system to lose the current correct state. The system moves in the state `UNDEFINED_P` until a signal re-setting the position of the level crossing is received. Under the assumption that a fault may occur at any time, an output edge labelled by `?f_p` and an output edge labelled `?f_t` exists starting from each state of the LTS.

The failure mode of the system may depend on the point of the execution of the system at which the fault occurs. In most cases, associating a fault action with a different failure mode to every state of the system is not necessary. Knowledge of the actual failure points and failure modes can be used to produce a simpler specification. Some examples in this direction are:

1. confining faults to specific subsystems

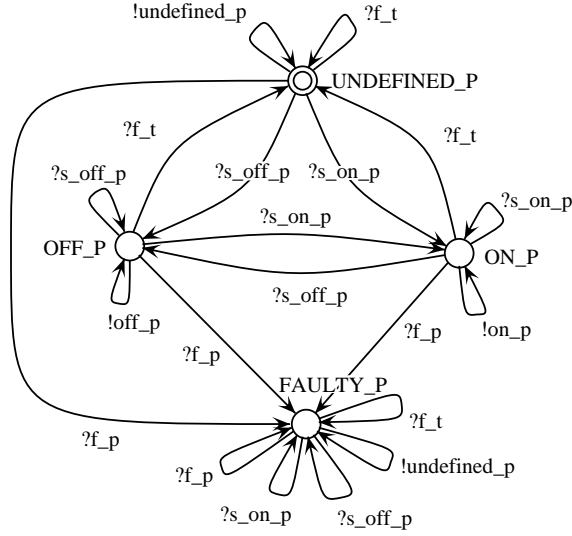


Figure 7: The failing system `gate_contr_p`

2. choose specific points in the execution of the subsystems at which a fault may occur, realising some form of guided fault injection
3. associating faults to communications between subsystems
4. assuming that every subsystem exhibits always the same failure mode in every state

A way to express the occurrence of a fault f at any point of the computation of the system P is the following:

$$(P' \parallel Q) \setminus f$$

where

- P' is equal to P except that every state of P is extended with the possibility of the occurrence of the fault followed by stop ($f : \mathbf{stop}$)
- Q is a process of the form $Q = f : Q'$ where Q' specifies the failure mode of the original system P

For every fault occurrence allowed by the fault assumption, a process Q must be instantiated.

Some process algebras, like LOTOS [5], include the *disabling* operator ($[>]$). The term $P[> (f; Q)$ means that the process P can be interrupted at any point by the action f . In this case the execution proceeds as Q . This operator allows the possibility of a fault occurring in every state to be expressed more concisely. However, this operator does not allow faults that can occur only in some states and not in other states to be modeled.

The modelling of faults that cause all subprocesses within a system to fail synchronously can be obtained by using the multiway synchronisation operator provided by the graphical notation. The port corresponding to a given fault in each replica is linked to the web operator. As an example, Figure 6 models the synchronous failure of the subsystems Q and R when the fault f occurs according to the fault assumption P .

Since the formalisms used in our approach see actions as atomic, the actions of the specification are atomic w.r.t. faults. If the actions of the specification model

functional activities of the real system, it may be needed to model instead faults that can occur *during* these actions. In this case, a different model of the behaviour of the system should be produced, for example, dividing functional actions in more atomic sub-actions and associating a choice of a fault action to each sub-action.

2.3 Introducing fault tolerance

The use of the parallel composition, restriction and relabelling operators of CCS/Meije (or graphical composition) is generally required in order to conveniently express a fault tolerant system design. A fault tolerance technique uses replicas of the system composed together with some extra standard components (for example, a majority voter) for masking the effects of the occurrence of faults. Formally, each replica is an instantiation of the failing system with an ad hoc renaming of actions. In particular, different names for the fault actions are used to distinguish between occurrences of the same kind of fault in different replicas.

Figure 8 shows the graphical specification of a classical *duplication and comparison* architecture applied to the gate example, duplicating the `gate_contr_p` process and adding a comparator process. The same figure shows that some actions must have the same name in all the replicas, while other actions must be renamed. The "set" signal must be sent synchronously to all replicas. The action `?s_on_p` needs not to be renamed in the replicas, since this action is actually a synchronisation action among the replicas. The actions `?f_p` must be instead renamed in all replicas, since this fault event is an asynchronous event for all of them.

Let n denote the number of replicas used by the fault tolerance technique and \mathcal{F}^j denote the set of faults of the j -th replica, $j = 1, \dots, n$. The set of faults of the fault tolerant system is therefore $\mathcal{F} = \bigcup_{j=1}^n \mathcal{F}^j$. Let $M = \{M_i, 1 \leq i \leq k\}$ be the set of extra components added by the fault tolerance technique (M may be empty).

The application of a fault tolerance technique leads to a network of replicated processes which includes the replicas and the added components synchronising in the specific way imposed by the fault tolerance technique. This is described in the CCS/Meije notation as (the parallel operator is left associative):

$$(\xi_1 \parallel \dots \parallel \xi_n \parallel M_1 \parallel \dots \parallel M_k) \setminus a_1, \dots, \setminus a_s$$

where a_1, \dots, a_s are the synchronisation actions, $a_i \notin \mathcal{F}$, and ξ_i is the i -th replica, with an appropriate renaming of the actions as explained above.

Since each replica is a distinct process, the specification of fault tolerance techniques based on design diversity is allowed. In this case instead of replicas, variants are used, each of which corresponds to a particular specification of the system.

Finally, error processing is generally achieved through error detection and recovery techniques. In this case, the error detection module can be specified as a further process which interacts with the failing system, checking states of the computation. Different actions can be used to distinguish various classes of errors, and the chosen error recovery algorithm can be modelled in the specification in a similar way.

2.4 Modelling fault assumptions

Assumptions on how faults are supposed to occur in the system can be specified by a further process, the *fault assumption process*, that is added to the specification by the parallel composition operator with synchronisation on the actions corresponding to faults. The fault assumption generally limits the number of fault occurrences. The most general fault assumption models any possible occurrence of faults. In the case of two faults, for example `f_p` and `f_t` above, this fault assumption is shown in Figure 9. The fault assumption (FA process) in Figure 8 allows the occurrence of at most one permanent fault in one of the replicas. In the initial state, either `!f_p_1` or `!f_p_2` can be executed. Then the process stops. Note that the label `FP1` on the

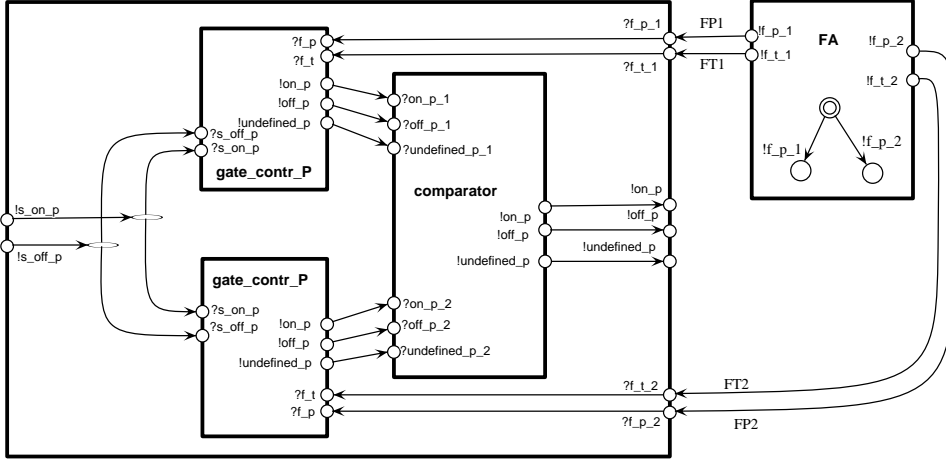


Figure 8: Fault tolerant system

link connecting the ports $?f_p_1$ and $!f_p_1$ produces an LTS in which, when this synchronisation action is executed, the action $FP1$ is observed. This label can be useful in the formalisation of properties about the behaviour of the system after the occurrence of the given fault.

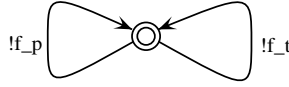


Figure 9: A fault assumption

3 Properties of fault tolerant systems

3.1 The logic ACTL

ACTL (Action-based Computation Tree Logic) [17] is an action-based version of the branching time temporal logic CTL [14]. ACTL has the advantage that, since it is based on actions rather than states, it is naturally interpreted over LTSs. Moreover, this logic is more expressive than other action-based logics, like Hennessy-Milner logic [24], without resorting to the full use of fixed point operators, such as the μ -calculus logic [29]. μ -calculus is more expressive than ACTL, but still most interesting properties can be expressed in the latter.

The formulae of ACTL are built over the syntactic categories of *action formulae*, *state formulae* and *path formulae*. An action formula permits expressing constraints on the actions that can be observed. A state formula gives a characterisation about the possible ways an execution can proceed after a state has been reached. A path formula states properties of an execution. The truth or falsity of a formula refers to a satisfiability relation over LTSs, denoted \models .

Given a set of observable actions Act , the action formulae on Act are defined as follows (α ranges over Act):

$$\chi ::= true \mid \alpha \mid \neg\chi \mid \chi \vee \chi$$

The satisfaction relation \models for action formulae is given by:

$$\begin{aligned}
\alpha &\models \text{true} && \text{always;} \\
\alpha &\models \beta && \text{iff } \alpha = \beta; \\
\alpha &\models \neg\chi && \text{iff } \alpha \not\models \chi; \\
\alpha &\models \chi \vee \chi' && \text{iff } \alpha \models \chi \text{ or } \alpha \models \chi'.
\end{aligned}$$

From now on, *false* abbreviates the action formula $\neg\text{true}$ and $\chi \wedge \chi'$ abbreviates the action formula $\neg(\neg\chi \vee \neg\chi')$.

The syntax of state formulae and path formulae is given by the grammar below:

$$\begin{aligned}
\phi &::= \text{true} \mid \neg\phi \mid \phi \& \phi' \mid E\gamma \mid A\gamma \mid <\chi>\phi \mid [\chi]\phi \\
\gamma &::= F\phi \mid G\phi \mid \phi\{\chi\}U\{\chi'\}\phi'
\end{aligned}$$

where χ, χ' range over action formulae, E and A are path quantifiers, F is the *eventually* operator, G is the *always* operator and U is the *until* operator.

The satisfaction relation \models for a state formula ϕ (path formula γ) by a state q (path ρ) is given inductively by:

$$\begin{aligned}
q &\models \text{true} && \text{always} \\
q &\models \neg\phi && \text{iff } q \not\models \phi \\
q &\models \phi \& \phi' && \text{iff } q \models \phi \text{ and } q \models \phi' \\
q &\models E\gamma && \text{iff there exists a full path } \theta \text{ from } q \text{ such that } \theta \models \gamma \\
q &\models A\gamma && \text{iff for all full path } \theta \text{ from } q, \theta \models \gamma \\
\rho &\models <\chi>\phi && \text{iff there exists } \alpha, q' \text{ such that } (q, \alpha, q') \in \rightarrow, q' \models \phi \text{ and } \alpha \models \chi \\
\rho &\models [\chi]\phi && \text{iff for all } q' \text{ such that } (q, \alpha, q') \in \rightarrow, q' \models \phi \text{ and } \alpha \models \chi \\
\rho &\models F\phi && \text{iff there exists a state } q \text{ in } \rho \text{ such that } q \models \phi \\
\rho &\models G\phi && \text{iff for all states } q \text{ in } \rho, q \models \phi \\
\rho &\models \phi\{\chi\}U\{\chi'\}\phi' && \text{iff there exists } \theta = (q, \alpha, q')\theta' \text{ suffix of } \rho, \text{ such that} \\
&&& q' \models \phi', \alpha \models \chi', q \models \phi \text{ and for all } \eta = (r, b, r')\eta', \\
&&& \text{suffixes of } \rho, \text{ of which } \theta \text{ is a proper suffix,} \\
&&& \text{we have } r \models \phi \text{ and } (b \models \chi \text{ or } b = \tau)
\end{aligned}$$

The modality $<\chi>\phi$ means that there exists a next state of the path, reached with an action satisfying χ in which the formula ϕ holds; while $[\chi]\phi$ says that for all next states of the path, reached with an action satisfying χ , the formula ϕ holds. These modalities correspond to the diamond and box modalities of Hennessy-Milner logic³. The meaning of the indexed until modality $\phi\{\chi\}U\{\chi'\}\phi'$ is that any state of the path is reached with an action in $\chi \cup \{\tau\}$ and the state satisfies the formula ϕ until a state is reached with an action in χ' and the state satisfies the formula ϕ' . Finally, note that $G\phi$ can be derived as $\neg F\neg\phi$ and $[\chi]\phi$ can be derived as $\neg <\chi>\neg\phi$.

Examples of properties for the **gate_contr_p** system and their formalisation in ACTL are:

- The system, after having received the action **?s_on_p**, cannot execute the action **!undefined**
 $\phi_1 = AG[?\text{s_on_p}]\neg EF[!\text{undefined_p}]\text{true}$
- The system eventually executes the action **!on_p**
 $\phi_2 = AF <!\text{on_p}>\text{true}$

³In [17], the ACTL modalities $<\chi>\phi$ and $[\chi]\phi$ are actually defined instead to be the weak version of the diamond and box operators

3.2 Properties verification

The AMC model checker available in the verification environment JACK [6] accepts a finite state machine (LTS) and an ACTL formula, and checks whether the formula holds on the LTS. The generation of the LTS from a network of subsystems is performed by means of other tools of JACK.

The time complexity of traditional model checking algorithms, which are used by AMC, is linear in the size of the global LTS and in the size of the ACTL formula, with respect to the number of different subformulae that can be syntactically recognised in it.

The model checker can be applied for the verification of the properties ϕ_1 and ϕ_2 described in the previous sub-section over the gate system. Formula ϕ_1 is satisfied by the specification of the gate without faults (Figure 4). On the other hand, the same formula is false for the specification of the gate that may fail (Figure 7), since after a fault the gate moves into a state which allows the action `!undefined` to be executed. Formula ϕ_2 is obviously *false* for both specifications of the gate. In fact this formula is *false* for every path which does not include the `?s_on_p` action. For example, the LTS in Figure 4 contains an infinite path from the initial state such that the `!undefined_p` action is always executed.

3.3 Formalising fault tolerance properties

The ACTL expression of the general classes of properties regarding the ability of the system of tolerating faults (see Section 1) are:

- *Fault tolerance*
 $AG\phi_{Corr}$
 where ϕ_{Corr} expresses a correctness condition on a state (*an invariant*)
- *Fail-stop*
 $AG[fault]\phi_{Term}$
 where ϕ_{Term} expresses the termination of the system
- *Fail-silence*
 $AG[fault]\phi_{CorrOmiss}$
 where $\phi_{CorrOmiss}$ expresses the correctness, apart from omission failures
- *Fail-safe*
 $AG[fault]\neg\phi_{Unsafe}$
 where ϕ_{Unsafe} expresses all possible unsafe behaviours

The general expressions given above mostly use the form $AG[fault]\phi$, which predicate over what should be valid forever in the life of the system after the occurrence of a fault. These kind of properties are called *safety properties*. Safety properties are distinguished from the *liveness properties*. Liveness properties state that something good should eventually (or infinitely often) happen in the system.

Depending on the nature of the system, safety and/or liveness properties may be needed to express fault-tolerance properties.

Example of properties concerning the behaviour of the `gate_contr_p` system are:

- *Fault tolerance property.*
 The system, after having received a signal `?s_on_p`, cannot execute either the action `!off_p` or the action `!undefined_p` until a signal `?s_off_p` has been received.

$$AG[?s_on_p] \neg E[true\{\neg?s_off_p\}U\{!off_p \vee !undefined_p\}true]$$

Similarly, after having received a signal `?s_off_p`, the system is not able to execute the action `!on_p` or the action `!undefined_p` until a signal `?s_on_p` has been received.

$$AG[?s_off_p] \neg E[true\{\neg?s_on_p\}U\{!on_p \vee !undefined_p\}true]$$

Fail-safe property.

The system, after having received a signal `?s_on_p`, cannot execute the action `!off_p` until a signal `?s_off_p` has been received.

$$AG[?s_on_p] \neg E[true\{\neg?s_off_p\}U\{!off_p\}true]$$

Similarly, after having received a signal `?s_off_p`, the system cannot execute the action `!on_p` until a signal `?s_on_p` has been received.

$$AG[?s_off_p] \neg E[true\{\neg?s_on_p\}U\{!on_p\}true]$$

- *Liveness property.*

The system, after having executed the action `!off_p`, eventually executes the action `!on_p`.

$$[!off_p]AF[!on_p]true$$

The fault tolerance property states that if the gate is open, then on receiving a request about its current state, the gate answers open, while if the state is closed the gate answers closed.

The fail-safe property is weaker, since it states that if the gate is open, then on receiving a request about its current state, the gate answers open or undefined. Similarly, if the gate is closed, the gate answers closed or undefined.

The liveness property guarantees that a closed gate will be eventually open.

When applying the model checker with these formulae to the system in Figure 7, the results are that the system satisfies the fail-safe property, but not the fault tolerance one. Also the liveness property is not satisfied by this system.

It is well-known that by applying the duplication-with comparison technique, the fault tolerant design tolerates one faulty replica. This can be proved by model checking by considering ad hoc fault assumption processes. For example, the fault assumption (FA process) in Figure 8 limits the occurrence of faults to at most one permanent fault in one of the replicas. Model checking shows indeed that the LTS of the fault tolerant system design in Figure 8 satisfies the fault tolerance property above.

4 State space explosion problem

The main difficulty in using in practice model checking formal verification methods is due to the limits imposed by the state space size problem, that even challenges more advanced model checking tools. Systems composed of several subsystems can be associated to a finite state model with a number of states which is exponential in the number of the component subsystems. Moreover, systems which are highly dependent on data values share the same problem, producing a number of states exponential in the number of data variables.

In the following it is shown an estimate of the maximal state-space size based on the structural knowledge of the system, i.e. on the observation that the phased structure of fault tolerant systems and algorithms limits a priori the state explosion problem. Indeed, a system employing redundancy is composed of a number of identical modules which compute the same results. At the architectural level such modules are often independent processors. Each module usually has a phased structure. Moreover, the modules have to synchronise periodically in order to maintain their consistency, and the synchronisations are usually combined with some comparison or voting operation, aimed to detect or mask errors.

A common structure of such a system can be represented as a network of subsystems; each subsystem synchronises with the other ones at the end of each phase. This structure is shown in Figure 10 in the case of duplication redundancy. The picture abstracts from the details of the synchronisation protocol, and also from the nature of the initial state of each phase, which generally is a set of states. These abstractions do not affect however the generality of the following observations.

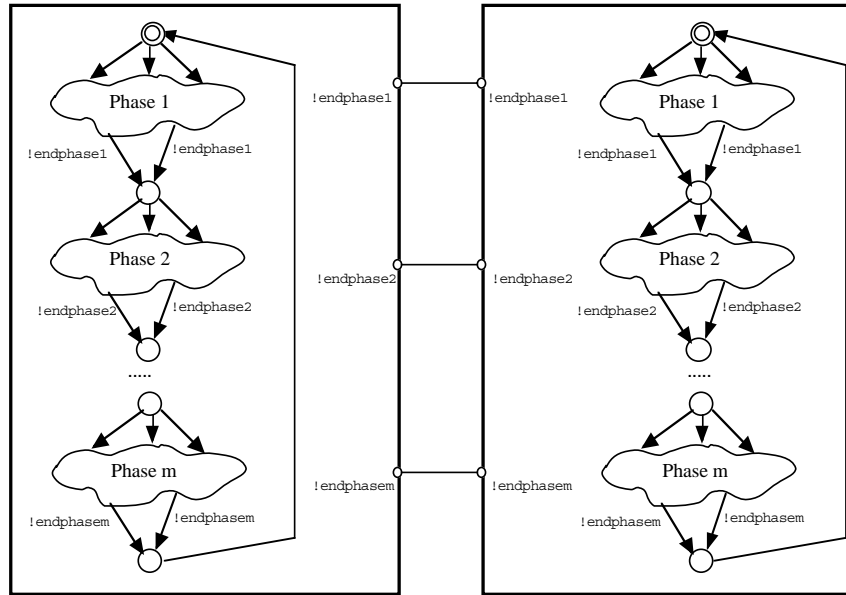


Figure 10: The phased structure

The behaviour of the overall system is obtained by the parallel composition of the replicas. Due to the synchronisation at the end of each phase, the obtained global LTS appears to be structured in phases as well; each phase of the overall system is actually generated by the interleaving of the corresponding phases of the different replicas, while each phase is terminated by the synchronisation of the replicas from which the next phase begins (see Figure 11, where $Phase\ i || Phase\ i$ represents the LTS built by interleaving two replicas of $Phase\ i$).

Let S be the size of the state space of a replica and S_i be the size of the state space of the i -th phase. The cardinality of the state space of the interleaving of n replicas has normally an upper bound of S^n . Due to the phased structure, the upper bound for S is determined by the size of the interleaving of each phase, that is: $S_1^n + S_2^n + \dots + S_m^n$.

Moreover, the regular structure of a redundant system may be exploited to contain state explosion with the help of existing established techniques, such as *symmetries* and *reduction preorders*. Using symmetries, as proposed by Emerson in [18], the number of states is reduced by identifying those states which coincide

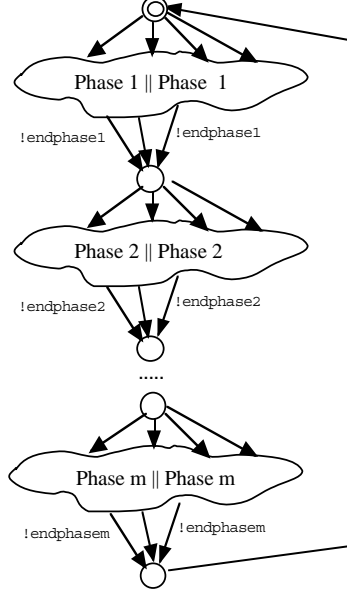


Figure 11: The phased structure

up to a permutation of the system components. Partial order reduction [21, 26, 42] employs the independency of the property to be checked from the order in which interleaved processes are actually executed, to select just one order and hence only a subset of the state space to check its validity. In the case of redundancy, the complete interleaving of the replicas can be avoided in the generation of the model. For example, the selected order of executions could be such that all the transition of the first replica precedes in each phase the transitions of the second replica and so on. The selection has however to take into account the interactions between the replicas. The global state space of a phase i of the global LTS for a system of n replicas is estimated to be of the order of $n * S_i$, and therefore the global state space of the overall algorithm is estimated to $n * S$.

Another observation that can be made is that the fault assumption process helps in the containment of state space explosion. Consider for example a case in which a replica is modelled by a sequence of phases, and in each of these phases, say the i -th, N_i states reachable in absence of faults and F_i states reachable within a failure mode are recognised (in reference to previous notations, it is $S_i = N_i + F_i$). If only a single fault is allowed to occur, say at the j -th phase, the total number of states is bound by the sum: $N_1^n + \dots + N_{j-1}^n + F_j * N_j^{n-1} + \dots + F_n * N_n^{n-1}$ (this formula refers to the case in which partial order reduction methods are not used).

The observations above are related to the redundant structure of the system. Other techniques can be used as well if the state space is still too large. Generally applicable techniques are decomposition and abstraction. For example, the following techniques can be applied:

1. Identification of the static configuration parameters of the system. The modelling of these attributes as if they were variables, contributes unnecessarily to the growth of the number of states of the model. In the development of the formal specification, the configurations can be taken each at a time and a property is satisfied by the system if and only if it is verified in all possible configurations
2. The relabelling of multiple actions into one action is a well-known reduction

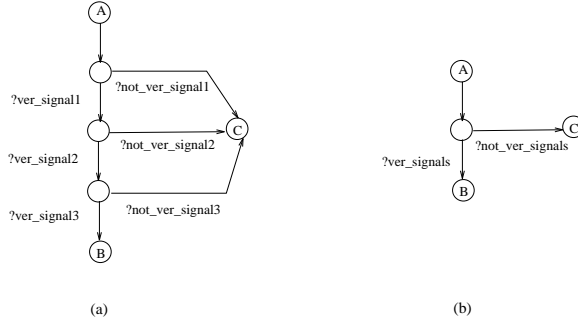


Figure 12: (a) A LTS. (b) The LTS after the reduction.

technique. For example, consider the LTS in figure 12 (a):

- the process sequentially tests several signals in order to execute an operation with success
- the failure of any of these tests leads to the failure of the operation itself
- the properties do not involve actions related to the tested signals

In this case the actions corresponding to a sequence of tests can be modelled as a non deterministic choice between the success and the failure of the tests, as shown in Figure 12 (b))

5 Case studies

This section shows an experience in the use of the proposed approach and verification tools on real case studies. The first study is the specification and verification of the safety requirements of a *Railway Interlocking System* developed by Ansaldo Trasporti [2]. The second one is the specification and verification of the Inter-consistency fault tolerant mechanism defined inside the project *GUARDS* (Generic Upgradable Architecture for Real-Time Dependable Systems) [37]. Both studies show that:

- the use of finite state machine as specification language has the advantage of ensuring the adherence of the produced formal specification to the original semi-formal one
- some standard rules for the passage from the semi-formal description of the system to its formal specification can be successfully applied in the field of fault tolerant systems. This passage is generally recognised one of the critical points of the introduction of formal methods in the software development cycle
- the reduction in the state space due to the phased structure of these kind of systems makes the model checking approach viable in this domain of application

5.1 Railway Interlocking System

The first case study is part of a railway signaling interlocking control system developed by Ansaldo Segnalamento Ferroviario. The system operates within a complex environment, interacting with human operators as well as with a number of different actuators and sensors. Sensors convey data concerning the physical status of

the environment, actuators allow for the control of the operations and the status of the external environment. An operator may interact with the system sending commands and selecting operation modes. The scope of this control system is that of permitting a safe passage of trains by adjusting the setting of signals on the railway line. The reader can refer to [2] for a detailed description of the specification and verification activity.

The control system is represented by a set of communicating processes, modelling logical and physical entities. The control of the entities is realised by operations which act on variables. Variables are easily modeled as processes, since they represent signals whose domain of values is very limited or signals for which a limited number of values are of interest. Finally, each operation is transformed into a process whose LTS describes the behaviour of the operation.

Every operation in the Ansaldo semi-formal specification has three main parts:

- the pre-conditions on variables that must be satisfied before continuing the operation (“VERIFY THAT” part)
- the execution of the operation, performed by modifying the value of some common variables (“ASSIGN” part)
- an “EXCEPTIONS” that specifies what should be done if a “VERIFY THAT” condition is not satisfied

An example of the description of an operation in the semi-formal specification is shown in Figure 13.

```
Automatic closure request
I. VERIFY THAT
  a. the command_state variable has the value "automatic";
  b. the lcc_state variable has a value not equal to
     "request to close".
II. ASSIGN
  - the value "manual" to the command_state variable
EXCEPTIONS
|a| |b|  command is lost; no recovery actions.
```

Figure 13: Semi-formal specification

The translation from the semi-formal to the formal CCS/Meije specification of the operation was straightforward. Figure 14 shows the specification of the operation in Figure 13.

5.1.1 Reduction of the number of states

The abstraction technique presented in the previous section for testing signal values was applied to every operation. The global LTS of the behaviour of the system resulted in about one million of states. The identification of the static configuration parameters of the system allowed a reduction in the number of states of this LTS to 77294 states.

5.1.2 Safety properties verification

A typical safety property for the interlocking system is: if the *proceed signal* is sent to the train when entering a track containing a level crossing, then the level crossing is closed. This property can be expressed more precisely as follows: in any state of

```

let rec {
  S = ?start_op: VERIF_A
  and
  VERIF_A = ?automatic : VERIF_B +
            ?manual : EXC
  and
  VERIF_B = ?closure_req : EXC +
            ?open_req : ASSIGN
  and
  ASSIGN = !s_manual : F
  and EXC = tau : F
  and F = !end : S
} in S;

```

Figure 14: Formal specification

the model if the position of the level crossing is not equal to closed, then there is no execution in which the *proceed signal* is sent until the position of the level crossing is equal to closed.

This property can be formalised as the following ACTL formula:

$$AG[\neg !\text{off_pos}] \neg E[true\{\neg ?\text{s_off_pos}\}U\{!\text{raise_shunt_sign}\}true]]$$

where $!\text{raise_shunt_sign}$ corresponds to the *proceed signal*, $!\text{off_pos}$ corresponds the state closed for the level crossing and $?s_off_pos$ corresponds to the set off signal received by the level crossing gate.

The property above is a fail-safe property. It was checked to be true on the specification of the interlocking system using the JACK environment.

5.2 Inter-consistency mechanism

The GUARDS project [37] has produced a generic architecture for safety critical systems designed to be instantiated to support different critical applications. Model checking has been used in the project to validate the Inter-consistency mechanism which is the basis of other ad hoc defined fault tolerant mechanisms.

The Inter-consistency mechanism must guarantee interactive consistency among three or four processors in the GUARDS architecture. Interactive consistency focuses on the problem of reaching agreement among multiple processors in presence of faults (also known as the "Byzantine Generals problem" [31]). The main difficulty to be overcome in achieving consistency is the possibility of conflicting values sent by faulty processors. Message authentication is assumed. This requires that faulty processors do not make undetectable modifications to messages as they are relayed from one processor to another.

The mechanism uses the ZA Byzantine Agreement algorithm described in [22]. This algorithm is synchronous and uses several rounds of authenticated encoded message exchange during which processor P tells processor Q what value it has received from processor R and so on. Each node has at the end a voted knowledge on each value hold by every other node. In [22] the generic algorithm has been verified using theorem proving techniques; the verification activity carried on inside the GUARDS project was instead aimed at the validation of the instances of the algorithm defined for the particular architecture. In the formal specification, two protocols have been identified: the transmitter and the receiver protocol. Every node has been specified as the composition of the protocols above. For example, in

phase 1:	phase4:
vp:p := p_encode(vp);	p2 := p_decode(msg2);
p_broadcast(vp:p);	msg3 := q_receive();
phase2:	phase5:
msg1 := s_receive();	p3 := p_decode(msg3);
	vp(p) := vote(p1, p2, p3);
phase3:	
p1 := p_decode(msg1);	
msg2 := r_receive();	

Table 1: The ZA algorithm of transmitter node P

the four nodes case P, Q, R and S, each node includes one transmitter protocol and three receiver protocols. The pseudo-code for the transmitter node P is given in Table 1, where `vp` is the private value of the node P. The mechanism is modelled as a network of four communicating processes, each modelling one of the four nodes. Moreover, since the algorithm has a phased structure: each process modeling a node is described by a network of communicating processes modelling the different phases of the algorithm and the local variables.

The translation from the pseudo-code to the formal specification is straightforward. For example, assuming two different values 0 and 1, the process modelling the `phase 2` of node P is expressed by the following CCS/Meije term:

```

phase2P = {
RECEIVE = ?ssendp_encp_0 : !s_m1p_encp_0 : END +
           ?ssendp_encp_1 : !s_m1p_encp_1 : END +
           ?ssendp_omission : !s_m1p_omission : END
and
END = !startphase3 : stop
} in RECEIVE;

```

The node on receiving a message from S (or detecting an omission fault), saves the message into the variable named `m1p`. The action `XsendY_encZ_j` corresponds to a message containing `j` encoded by `Z` and sent by `X` to `Y`; `XsendY_omission` corresponds to an omission fault from `X` to `Y`; `s_m1p_encZ_j` corresponds to storing the value `j` encoded by `Z` into the variable `m1p`.

Then the node is ready to execute `phase 3` of the protocol, and signals this by the `!startphase3` action, on which all the other nodes have to synchronise. The complete specification and verification work is reported in [3].

5.2.1 Reduction of the number of states

Table 2 presents the size of the state space of the single node, and that of the network composed of four nodes under different fault assumptions. The fault assumptions have been modelled by means of specific processes which constrain the occurrences of faults.

The table shows that:

- the size of the state space of the network with four replicas is largely below the fourth power of the size of the state space of a single node
- the state space increases with the generality of the fault assumptions, as evident in the last two rows

Model of:	states
A single non faulty node	428
Network of 4 non faulty nodes	3479
Network with an arbitrarily faulty node and a symmetric faulty node	109613
Network with an arbitrarily faulty node, and authentication violation	122767

Table 2: Number of states for the GUARDS Byzantine Agreement.

5.2.2 Agreement and Validity properties verification

The classical *Agreement* and *Validity* properties must be satisfied to reach consistency:

Agreement: if a pair of receivers are non faulty, then they agree on the value ascribed to the transmitter.

Validity: if the receiver P is non faulty, then the value ascribed to the transmitter by P is the value actually sent if the transmitter is non faulty or symmetric faulty; or the distinguished value *error*, if the transmitter is manifest faulty.

Consider only two possible values (0 and 1). *Agreement* can be formalised as: for any execution, the non faulty nodes eventually agree on the value 1 or the nodes eventually agree on the value 0.

Assume P is non faulty. *Validity* can be formalised as: if in any state of the model, it is true that the internal value of the node P is equal to 1 or 0, then for any execution of the processes, starting from such a state, if the processes are non faulty, they eventually agree on such a value.

Assume *S* is faulty. The combination of the *Agreement* and *Validity* properties in the case of value 1, is expressed by the following ACTL formula:

$$AG[\neg \text{psend_vp_1}](A[\text{true}\{ \text{true} \} U \{ \neg \text{vp_ofp_eqto_1} \} \text{true}] \& \\ A[\text{true}\{ \text{true} \} U \{ \neg \text{vp_ofq_eqto_1} \} \text{true}] \& \\ A[\text{true}\{ \text{true} \} U \{ \neg \text{vp_ofr_eqto_1} \} \text{true}])$$

where the action $\neg \text{psend_vp_1}$ indicates that the private value of the node P is 1 and the action $\neg \text{vp_ofY_eqto_1}$ indicates that 1 is the value ascribed by the receiver node Y to the transmitter node P.

The properties above falls in the class of fault-tolerance properties, and model checking has been applied to prove their invariance under different fault assumptions. As expected, in the case of a violation of the assumption on authentication, even a single faulty node is not tolerated.

6 Related work

The approach presented in this paper applies model-checking to fault tolerant systems specified by using a standard process algebra. Faults are modeled as observable actions. The observability of faults is not related to the possibility of detection of faults (fault detection mechanisms usually detect the consequences, rather than the fault occurrence itself). It enables to clearly distinguish faults from other internal actions, and to control fault events, so that fault assumptions modeling is possible.

In the literature on the formalisation of fault tolerant systems, the earlier works ([15, 41]) do not model explicitly the occurrence of faults, but only the failure behaviour.

Several later works are based on the use of standard process algebras to specify the behaviour of the system also under fault occurrences; equivalence relations or preorders are employed to verify fault tolerant system designs. The major advantage of standard process-algebra based approaches is related to the existence of automatic verification tools. In particular, CCS process algebra and its observational equivalence [34] has been first used for this purpose in [38]. In [19], CCS is used in verifying a distributed control for a railway block signaling system. In [40] and [35] CSP and its trace theory [25] are applied. In [36], CSP and assertional techniques are combined to design fault tolerant systems based on dynamic redundancy; refinement steps and proof obligations are applied.

Other works in the literature use instead specialised process algebras: in [30], a CCS-like calculus for replicated systems is presented. In [16], new process algebra operators to model faults and failure modes are defined. In [28, 27] a new semantics for CCS is defined, which is parameterised on the fault assumptions.

Verification of system fault tolerance properties has also been addressed both with theorem proving and model checking techniques.

Theorem proving has been applied to study fault tolerance in [22]. The specification language is a strongly typed high-order logic, and the PVS theorem prover allows semi-automatic proofs to be generated.

In [32], a calculus for fault tolerance analysis based on TLA, the Temporal Logic of Actions, is defined. Theorems asserted in the specification are proved using the method of structured proofs.

In [23] the micro-CRL and a modal logic for this language are used for modelling a railway interlocking system and their safety properties. Properties are then verified by transforming the specification in propositional logic and by using a theorem prover.

Model checking of properties expressed in modal μ -calculus on CCS specifications is first applied in [11, 9] to the verification of fault properties of a railway interlocking system. In [10] fault-handling mechanisms are modelled using special-purpose process operators; temporal properties which hold for fault tolerant mechanisms applied to simple processes are shown to hold as well when the mechanisms are applied to more complex processes. The use of modal transition systems is exploited in [8], where a modal process logic that captures the intention behind failures is defined.

The model checking approach presented in this paper is based on traditional process algebras in order to be able to exploit the powerful verification capabilities offered by existing verification environments. Moreover, the modelling of faults as observable actions, allows the verification under different fault scenarios.

7 Conclusions

This paper shows the application of the model checking technique for the specification and verification of fault tolerant systems. The results on the application of the approach to two case studies are reported. The studies show the feasibility of model checking to real examples, and confirm that key-point in the acceptance of model checking are the use of a specification formalism which is essentially some variants of finite-state machines (commonly used in many industrial activities, especially in the safety critical systems area) and the existence of automatic verification tools.

State explosion represents the main problem to the application of model checking for handling large systems. However, recent advances in model checking techniques have managed to deal with very large state spaces by the use of symbolic manipulation algorithms inside model checkers. The most notable example is the SMV model checker [12]. In SMV the transition relations are represented implicitly by means

of Boolean formulae and are implemented by means of Binary Decision Diagrams (BDDs, [7]). This usually results in a much smaller representation for the systems' transition relations, thus allowing the maximum size of the systems that can be dealt with to be significantly enlarged. These advances, together with what reported in this paper about the state space of redundant systems, indicate that fault tolerant systems are a promising field of application of model checking techniques.

Acknowledgments

This work has been partially supported by the Italian Ministry of University and Research within the projects COFIN QUACK and 5%SP4.

Special thanks go to the anonymous reviewers for their appropriate remarks and their suggestions.

References

- [1] Austry D, Boudol G. 1984. Algebre de processus at synchronisation. *Theoretical Computer Science*, **1**(30): 91-131.
- [2] Bernardeschi C, Fantechi A, Gnesi S, Larosa S, Mongardi G, Romano D. 1998. A formal verification environment for railway signaling system design. *Formal Methods in System Design*, **12** : 139-161.
- [3] Bernardeschi C, Fantechi A, Gnesi S. 1999. Formal validation of fault tolerance mechanisms inside GUARDS. *Proceedings SAFECOMP'99*, Toulouse, Lecture Notes in Computer Science **1698**: 420-430.
- [4] Bernardeschi C, Fantechi A, Simoncini L. 2000. Formally verifying fault tolerant system designs. *The Computer Journal*, **43**(3): 191-205.
- [5] Bolognesi, T, Brinksma E. Introduction to ISO Specification Language LOTOS. *Comp. Networks and ISDN Systems*, **14**: 25-59.
- [6] Bouali A, Gnesi S, Larosa S. 1994. The integration project for the JACK environment. *Bulletin of the EATCS*, **54**: 207-223.
- [7] Bryant RE. 1986. Graph based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, **C-35**(8).
- [8] Bruns G. 1995. Refinement and dependable systems. *Proceedings 10th Annual Conference on Computer Assurance*, IEEE : 49-55.
- [9] Bruns G. 1997. *Distributed systems analysis with CCS*. Prentice Hall.
- [10] Bruns G, Sutherland I. 1997. Model checking and fault tolerance. *Proceedings 6-th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science **1349**, Sydney, Australia: 45-59.
- [11] Bruns G. 1992. A case study in safety-critical design. *Proceedings Computer Aided Verification '92*, Lecture Notes in Computer Science **663**: 220-234.
- [12] Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, **98**(2): 142-170.
- [13] *Railway Applications: Software for Railway Control and Protection Systems*. 1994. European Committee for the Electrotechnical Standardization (CENELEC), EN 50128.

- [14] Clarke EM, Emerson EA, Sistla AP. 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, **8**(2): 244-263.
- [15] Cristian F. 1985. A rigorous approach to fault tolerant programming. *IEEE Transaction on Software Engineering*, **11**(1): 23-31.
- [16] De Boer FS, Coenen J, Gerth R. 1993. Exception Handling in Process Algebra. *Proceedings First North American Process Algebra Workshop*, Workshop in Computing Series, Springer-Verlag.
- [17] De Nicola R, Vaandrager FW. 1990. Actions versus state based logics for transition systems. *Proceedings Ecole de Printemps on Semantics of Concurrency*, Lecture Notes in Computer Science **469**, Springer, Berlin : 407-419.
- [18] Emerson EA, Sistla AP. 1993. Symmetry and model checking. *Proceedings Computer Aided Verification'93*, Lecture Notes in Computer Science **697**, Springer, Berlin.
- [19] Fisher S, Scholz A, Taubner D. 1992. Verification in process algebra of the distributed control track vehicles-A case study. *Proceedings Computer Aided Verification '92*, Lecture Notes in Computer Science **663** : 192-205.
- [20] Formal Methods Europe Applications Database. <http://www.fmeurope.org/databases/full.html>
- [21] Godefroid P. 1990. Using partial orders to improve automatic verification methods *Proceedings Computer Aided Verification '90*, Lecture Notes in Computer Science **531**: 176-185.
- [22] Gong L, Lincoln P, Rushby J. 1995. Byzantine agreement with authentication: observations and applications in tolerating hybrid and link faults. *Proceedings 5th Conference on Dependable Computing for Critical Applications*, (DCCA-5), Urbana-Champaign, USA.
- [23] Groote JF, van Vlijmen SFM, Koorn JWC. 1995. The safety guaranteeing system at station Hoorn-Kersenboogerd. *Proceedings 10th Annual Conference on Computer Assurance*, IEEE : 57-68.
- [24] Hennessy M, Milner R. 1985. Algebraic laws for nondeterminism and concurrency. *ACM*, **32**(1): 137-161.
- [25] Hoare, C.A.R. 1985. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs.
- [26] Holzmann GJ, Peled D. 1994. An improvement in formal verification. *Proceedings FORTE 1994 Conference*, Bern, Switzerland.
- [27] Janowski T. 1997. On bisimulation, fault-monotonicity and provable fault-tolerance. *Proceedings 6-th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science **1349** : 292-306.
- [28] Janowski T. 1994. Fault-tolerant bisimulation and process transformations. *Proceedings 3-rd International Symposium on Formal techniques in Real-Time and Fault Tolerant Systems*, Lecture Notes in Computer Science **863** : 372-392.
- [29] Kozen D. 1983. Results on the propositional mu-calculus. *Theoretical Computer Science*, **27** : 333-354.

- [30] Krishnan P. 1994. A semantic characterisation for faults in replicated systems. *Theoretical Computer Science* **128**: 159-177.
- [31] Lamport L, Shostak R, Pease M. 1982. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3): 382-401.
- [32] Lamport L, Merz S. 1994. Specifying and verifying fault-tolerant systems. *Proc. 3-rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, Lecture Notes in Computer Science **863**: 41-76.
- [33] Laprie JC (Ed.). 1992. *Dependability: basic concepts and terminology*. Dependable Computing and Fault-Tolerant Systems, **5**, Springer-Verlag.
- [34] Milner R. 1989. *Communication and concurrency*. Prentice-Hall International, Englewood Cliffs.
- [35] Nordahl J. 1992. Design for dependability. In Landwehr, C.E., Randell, B., Simoncini, L. (eds), *Dependable Computing for Critical Applications 3*, Dependable Computing and Fault-Tolerant Systems series, **8**, Springer-Verlag: 65-89.
- [36] Peleska J. 1990. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, **5** (2): 95-106.
- [37] Powell D, Arlat J, Beus-Dukic L, Bondavalli A, Coppola P, Fantechi A, Jenn E, Rabejac C, Wellings A. 1999. GUARDS: a generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, **10**(6): 580-599.
- [38] Prasad KVS. 1984. Specification and proof of a simple fault tolerant system in CCS. *Internal Report CSR-178-84*, Dept. of Computer Science, Univ. of Edinburgh.
- [39] Roy V, De Simone R. 1990. AUTO and Autograph. *Proceedings of the Workshop on Computer Aided Verification*, Lecture Notes in Computer Science **531**: 65-75.
- [40] Schepers H, Hooman J. 1994. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science* **128**: 127-157.
- [41] Schneider FB. 1990. Implementing fault tolerant services using the state machine approach: a Tutorial. *ACM Computing Surveys* **22** (4): 299-319.
- [42] Valmari A. 1990. A stubborn attack on state explosion *Proceedings Computer Aided Verification '90*, Lecture Notes in Computer Science **531**: 156-165.