

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 shows how fault tolerant systems can be analysed by model checking formal verification technique. A modelling approach suitable for model checking fault tolerant systems under different fault scenarios is presented. The approach is included in a general development framework that has been proved to be usable for the verification of a railway interlocking system and fault tolerant mechanisms.

1 Introduction

The large deployment of computer-controlled systems has raised many concerns are raised about safety issues when human activities and lifes depend on them. A combination of fault prevention, fault tolerance, fault removal and fault forecasting techniques are commonly used in order to achieve high degree of dependability. There not exists, however, a common agreement about a standard method to orderly combine these different techniques. Industries, also basing on their different backgrounds and application fields, adopt different development trajectories, and the various techniques aimed at enhancing dependability are normally separately used. Indeed, the combination and integration of so different techniques is still an open research area.

In this paper, we address the combination of the provision, in the development of a system, of fault tolerance mechanisms and the use of formal methods, and in particular formal verification tools. 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 [5], 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 fault tolerant 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

*This work was partly supported by Progetto Coordinato CNR "Strumenti Automatici per la Verifica Formale nel Progetto di Sistemi Software".

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, we propose in this paper the use of a formal verification technique, namely model checking, to verify the conformance of a design with respect to given fault-tolerance properties, regarding its ability to tolerate faults, such as:

1. *Correctness*. The system delivers a correct service (in absence of faults).
2. *Fault tolerance*. Despite faults, the system delivers a correct service.
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 of 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 done, 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 [15].

In this paper an approach in the application of model checking to the study of fault tolerance properties is presented, showing also an experience in the use of verification tools on case studies concerning real systems, and in addressing the "State Space Explosion" problem that can arise when a system is composed of several subsystems. 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. At a first sight the presence of redundancy, which is often introduced by fault tolerance mechanisms, seems to raise the state space explosion problem since it increases, often duplicating or triplicating, the number of subsystems. In this paper it is argued that this is not in general true, and that instead some typical redundant structures can help to contain the state space, and the most suitable techniques to adopt in order to address this problem are indicated.

The paper is organized as follows. Section 2 reviews related work. Section 3 presents a technique adopted for formally specifying fault tolerant systems. Section 4 reviews fault tolerant systems properties, their formalisation and verification. Section 5 deals with the characteristics that make model checking techniques applicable in this field. Section 6 reports on the application of the proposed formalization techniques and verification tools to two case studies of fault tolerant systems and their properties. Section 7 concludes the work.

2 Related work

In the literature on the formalisation of fault tolerant systems, the earlier works ([16, 46]) 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. CCS process algebra and observational equivalence has been first used in [43]. In [23], CCS process algebra is used in verifying a distributed control for railway block signalling system. In [45] and [39] CSP and trace theory is applied. In [41], CSP and assertional techniques are combined to design fault tolerant systems based on dynamic redundancy. Refinement steps and proof obligations are applied. The major advantage of process-algebra based approaches is related to the existence of automatic verification tools.

Other works in the literature use instead specialised process algebras: in [32], a CCS-like calculus for replicated systems is presented. In [18], new process algebra operators to model faults and failure modes are defined. In [30, 29] a new semantics for CCS is defined, which is parameterised on the fault assumptions.

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

The theorem proving technique has been applied to study fault tolerance in [25]. The specification language is strongly typed high-order logic, and a theorem prover allows to generate semi-automatic proofs.

In [34], 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. Simple proofs can be checked mechanically through the TLP verification system.

In [26] 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] 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. Finally, [9] is a book on the use of CCS for distributed systems analysis.

The approach presented in this paper applies model-checking to fault tolerant systems defined using a standard process algebra. Faults are modeled as observable actions: the observability of fault is not related to the possibility of detection of faults (fault detection mechanisms usually detect the consequences, rather than the fault occurrence itself), but rather they enable to clearly distinguish faults from other internal actions, and to control fault events, so that fault assumptions modeling is possible. The use of standard process algebras allows already developed verification tools to be used.

3 Modelling Fault tolerant Systems

This section presents an approach to specify fault tolerant systems so that the specification can be analysed by model checking technique. The following concepts and terminology are used:

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. Faults are modeled as random events. For example, a crash fault in a state extends the behaviour of the system by allowing a crash to occur in that state.
- 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.

3.1 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 the subset of Meije process algebra, defined in [1], that corresponds to the CCS process algebra, following R. Milner [38].

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:

$$\begin{aligned}
 R &::= \mathbf{stop} \mid X \mid a : R \mid R + R \mid \\
 &\quad \mathbf{let\ rec} \{X = R \mid \mathbf{and} \ X = R\} \mathbf{in} \ X \\
 P &::= R \mid P \parallel P \mid P \setminus \alpha \mid P[\alpha/\beta] \mid \\
 &\quad \mathbf{let} \{X = P \mid \mathbf{and} \ X = R\} \mathbf{in} \ X
 \end{aligned}$$

where

- 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 without behavior
- $a : R$ 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
- $R + R$ is the non deterministic choice operator: a process can choose between the behaviour of several processes
- 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. In particular, only finite state LTSs are considered here, since the two layered syntax of CCS/Meije allows only finite state processes to be defined¹.

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 $\rho \neq q$.

The Figure 1 shows the structural operational semantics of some CCS/Meije operators previously described, in terms of LTSs².

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)

¹the 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

²CCS/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.

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

and the process `close_p` (the closure procedure). The process `gate_contr_p` has three states: `undefined` (initial state), `on` (gate open) and `off` (gate closed). The gate changes its state upon receiving a command from the other processes. The open operation checks the state of the gate. If the state is different from `on`, it sends to the gate the `s_on_p` signal. Similarly, the closure operation checks the state of the gate. If the state is different from `off`, it sends to the gate the `s_off_p` signal.

Act contains the following actions: `on_p`, `off_p` and `undefined_p`, executed by the `gate_contr_p` to signal its current state; `start_close_op` and `end_close_op`, executed by the process `close_op` when this operation begins/ends; `start_open_op` and `end_open_op`, executed by the process `open_op` when this operation begins/ends; `s_on_p`, sent by the process `open_p` to the process `gate_contr_p` for setting the state of the gate to `on`; `s_off_p`, sent by the process `close_p` to the process `gate_contr_p` for setting the state of the gate to `off`. These last two actions are synchronisation actions.

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 either execute the action `!on_p` indicating the current state of the process or receive a signal `!s_off_p` (set state off) and changing its 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 brevity the specifications of the `open_op` and `close_op` processes (which would require more information on how the external environment commands the operations) 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 or when commanding the change of the level crossing state (actions: `on_p`, `off_p`, `undefined_p`, `s_off_p` and `s_on_p`).

```
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

The graphical notation, defined for the ATG tool [44], can be used. 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 process places of interconnection with the environment. If two boxes are drawn at the same level, they can synchronize via the actions they execute by linking the corresponding ports.

Figures 4 report the graphical specification of the `gate_contr_p` and the `net` systems, 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.

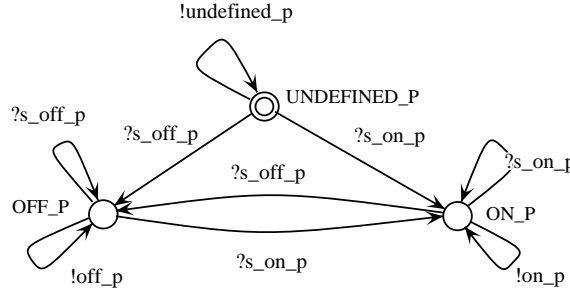


Figure 4: The `gate_contr_p` graphical specification.

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

- Observing 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

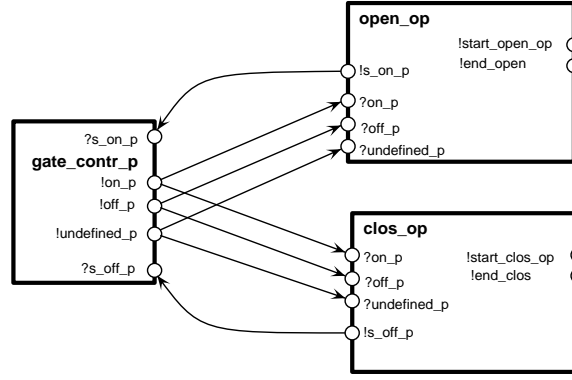


Figure 5: A network of processes

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 .

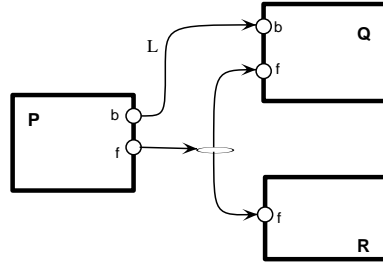


Figure 6: Two-way and multi-way synchronisation

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

3.2 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 faults in the system. The specification of the failure of the system is obtained by introducing occurrences of possible faults as transitions of the LTSs of 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 system shows forever the value **undefined** to the environment (the action **!undefined_p**). The state **FAULTY_P** is a sink state. The temporary fault causes the system to lose the current correct state, by showing the value **undefined** until the reception of a signal setting the position of the level crossing. Under the assumption that a fault may occur at any time, an output edge labelled by **?f_p** and an output edge labelled by **?f_t** exists starting from each state of the entity.

The failure mode of the process may depend on the point at which a fault occurs during the execution of the system. In most cases, associating a fault action with

Since the formalisms used in our approach see actions as atomic, the actions of the specification are atomic w.r.t. faults. In the case of modeling faults that can occur *during* the occurrence of a functional action, a different model of the behaviour of the system should be produced, where functional actions are divided in more sub-actions. A choice between a sub-action and a fault action is performed at each of them. The model of the functional behaviour of the system should be designed with a granularity that fits the sought granularity of fault occurrences.

3.3 Introducing fault tolerance

The formal modeling of a fault tolerant system can be often structured as the parallel composition of replicas that synchronise to produce useful work. The formalization of a fault tolerance technique requires the use of the parallel composition, restriction and relabelling operators of CCS/Meije (or graphical composition) in order to conveniently express composition of redundant replicas and additional components.

If fault masking is applied, a fault tolerance technique uses replicas of the system. Replicas are generally composed together with some extra standard components added by the fault tolerance technique (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 and different names for the fault actions (to distinguish between occurrences of the same kind of fault in different replicas).

For example Figure 8 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 among the replicas. The actions `f_p` must be renamed instead in all replicas, since this fault event is asynchronous 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 synchronizing in the specific way dictated by the fault tolerance technique (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. We assume processes corresponding to replica use appropriate renaming of the actions.

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

Figure 8 shows the specification of a classical *duplication and comparison* architecture applied to the gate example, duplicating the `gate_contr_p` process and adding a comparator process.

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; the recovery algorithm can be included in the specification of the failing system. Different actions can be used to distinguish various classes of errors, and the error recovery algorithm followed can be modelled in the specification in a similar way.

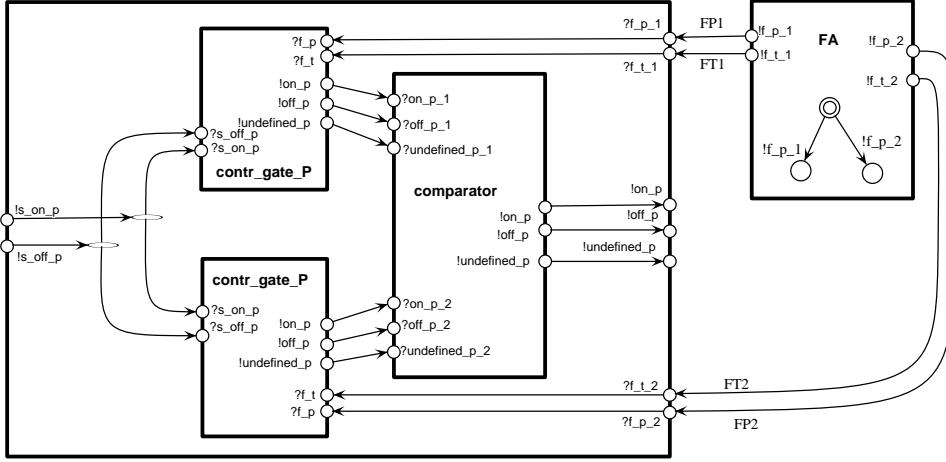


Figure 8: Fault tolerant system

3.4 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.

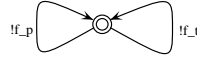


Figure 9: A fault assumption

The fault assumption in Figure 8 does not allow the occurrence of temporary faults and at most one permanent fault in one of the replicas can occur. Consider the FH process. In the initial state, either the $!f_p_1$ action is executed or the action $!f_p_2$ action is executed. Then the process stops.

4 Properties of fault tolerant systems

A temporal logic in agreement with the selected specification formalism is used to formalise system's properties.

4.1 The logic ACTL

ACTL (Action-based Computation Tree Logic) [19] is an action-based version of the branching time temporal logic CTL [15]. 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 [27], without resorting to the full use of fixed point operators, such as the μ -calculus logic [31]. 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 characterization 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):

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

An action formula permits expressing constraints on the actions that can be observed. The satisfaction relation \models for action formulae is defined as follows:

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

From now on, we let *false* abbreviate the action formula $\neg true$ and $\chi \wedge \chi'$ abbreviate the action formula $\neg(\neg\chi \vee \neg\chi')$.

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

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

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.

Satisfaction of a state formula ϕ (path formula γ) by a state q (path ρ), notation $q \models \phi$ (or just $\rho \models \gamma$), is given inductively by:

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

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 $\neg <\chi>\neg\phi$.

³in [19], the modalities $<\chi>\phi$ and $[\chi]\phi$ are actually defined instead to be the weak version of the diamond and box operators

Some properties for the `gate_contr.p` system in Figure 4 and their formalisation in ACTL are:

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

4.2 Properties verification

The model checker accepts a finite state machine (LTS) and an ACTL formula [22]. If the model checker determines the formula is true, then the property holds in the LTS and also in the system specification.

The time complexity of traditional model checking algorithms, which are used in the model checker of the JACK environment, is linear in the size of the global LTS and in the size of the ACTL formula (the number of different subformulae that can be syntactically recognized in it) to be checked.

The model checker provides also the *counterexample* facility. If we check that our specification has a certain property, using this facility we can discover the paths that make such a property true or false on the model.

Consider the failing `contr_gate.p` system and the properties ϕ_1 and ϕ_2 in Section 4.1. ϕ_1 is satisfied by the LTS in figure 4. ϕ_2 is obviously *false* for every path which does not include the `?s_on_p` action. We obtain the following trace from the counterexample facility of the model checker:

```
|= AF <!on_p> true
The formula is FALSE in state 3:
|= why
(<!"on_p"> true)
is false in state 3
UNDEFINED - !undefined_p - UNDEFINED
```

4.3 Formalising fault tolerance properties

The ACTL expression of the general classes of properties reported in 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 actual formulae to be checked strictly depends on the functionality of the system, as we will see in the next Section. However, 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 have often been called *safety properties* and are often satisfied by a “null” system.

Safety properties are distinguished from the *liveness properties*. Liveness properties are expressed by the forms $AF\phi$, $EF\phi$, $AGAF\phi$, and so on, and state that something should be eventually (or infinitely often) done by the system.

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

Example of properties (and their formalisation in ACTL) of the `gate_contr_p` system, whose LTS is shown in Figure 7, are:

- *Fail-safe property.*

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

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

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

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

- *Fault tolerance property.*

The system, after having received a set on signal `?s_on_p`, executes the action `!on_p` until a set off signal has been received.

Similarly, after having received a set off signal `?s_off_p`, the system executes the action `!off_p` until a set on signal has been received.

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

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

- *Liveness property.*

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

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

Fail-safe property guarantees that if the gate is open, then the state is on or undefined. Similarly when the gate is closed, then the state is off or undefined. This holds also in presence of faults.

Fault tolerant property states that if the gate is open, then the state is on, while if the state is closed the state is off. This holds also in presence of faults.

The liveness property guarantess that a closed gate eventually is open, this would be useful for the actual users of the level-crossing.

The system in Figure 7, satisfies fail-safety property, while it is not fault tolerant. Also the liveness property is not satisfied by the system.

The fault tolerant system design in Figure 8 instead tolerates one faulty replica. The fault assumption process in the same figure limits the occurrence of faults to at most one permanent fault in one of the replicas. This system satisfies the fault tolerance property. Assume the `FH` process in Figure 8 is replaced by the `FH` process in Figure 9. Since faults are not limited, in this case the fault tolerant property above is not satisfied.

5 State space explosion problem

The main difficulty in using in practice model checking formal verification methods is due to the limits imposed by the 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 structural knowledge about the system. The phased structure of fault tolerant systems and algorithms limits a priori the state explosion problem, even if adopting traditional model checking algorithms. A system employing redundancy is composed of a number of identical modules which compute the same results. At the architecture level such modules are often independent processors. These modules have to synchronize periodically in order to maintain their consistency, and the synchronizations 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 (in the case of duplication redundancy) as shown in Figure 10, as a network of automata; each LTS synchronizes with the other ones at the end of each phase.

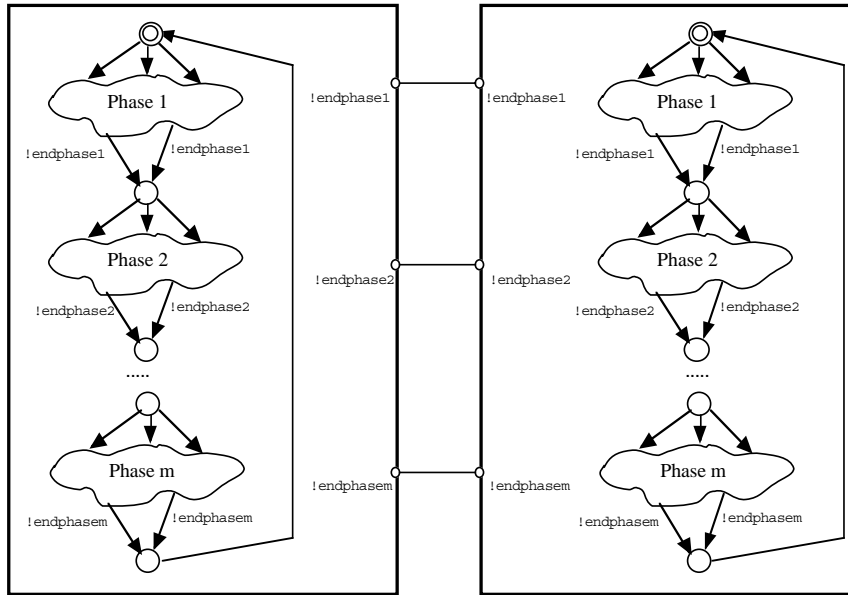


Figure 10: The phased structure

The behaviour of the overall system is obtained by the parallel composition of the replicas. Due to the synchronization 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 synchronization 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 $Phase\ i$).

If we call S the size of the state space of a replica, 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, if we denote by S_i the size of the state space of the i -th phase, 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 [21], the number of states is reduced by identifying those states which coincide

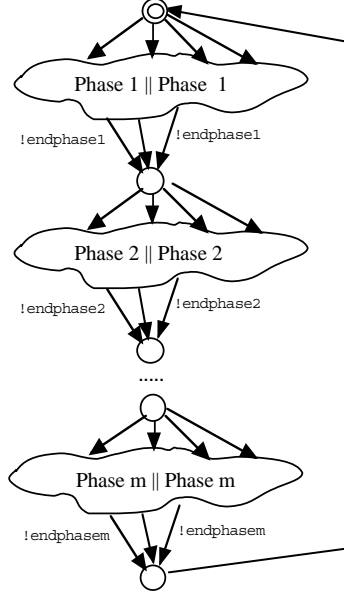


Figure 11: The phased structure

up to a permutation of the system components. Reduction preorders [28] employ 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 executions could satisfy the constraint that all the transition contributed to the global LTS by the first replica precedes 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 $n * S$.

Finally, the use of techniques such as decomposition and abstraction, can be applied to overcome the state space explosion problem at the specification level.

In particular, the following technique can be applied:

1. identification of the system's static configuration parameters. The modelling of these attributes as if they were variables, would contribute 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 reduction well-known technique. Assume 1) an entity sequentially tests several signals in order to execute an operation with success; 2) the failure of any of these tests leads to the failure of the operation itself; 3) 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. Consider the LTS in figure 12 (a). The failure of any test leads to state C, while the success of all the tests leads to state B. The sequential tests can be substituted by a simple abstract test which may fail,

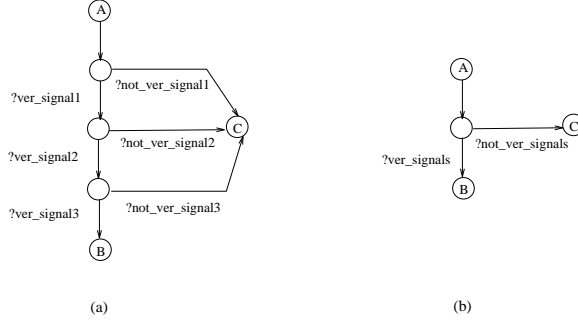


Figure 12: (a) A LTS. (b) The LTS after the reduction.

leading to state C; or it may be executed with success leading to state (Figure 12 (b)).

3. 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, we can recognize N_i states reachable in absence of faults and F_i states which are reached within a failure mode (in reference to previous notations, we have therefore that $S_i = N_i + F_i$). 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: $N_1^n + \dots + N_{j-1}^n + F_j * N_j^{n-1} \dots + F_n * N_n^{n-1}$, if we are not using reduction preorder. REFEREE 2; cite general work on partial order methods for model checking

In addition to the above techniques, domain-specific optimization of model checking algorithms have been studied in the literature. In particular, some specific features of safety critical systems may be searched that can be used to optimize the verification algorithms. As an example of possible domain specific optimizations, Eisner [20] has shown how the safety critical characteristics of robustness and locality can be used to avoid difficult fixed-point calculations in symbolic model checking when applied to railway interlocking.

6 Case studies

The approach proposed in the paper has been applied for specifying and verifying two fault tolerant system designs. 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 fault tolerant mechanisms defined inside the project *GUARDS* (Generic Upgradable Architecture for Real-Time Dependable Systems) [42]. Both studies show that:

- 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 recognized 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 redundant systems makes the model checking approach viable in this domain of application.

6.1 Railway Interlocking System

The first case study is a part of a railway signaling interlocking control system developed by Ansaldo Segnalamento Ferroviario. The system operates within a complex environment, interacting with a number of different actuators and sensors, and human operators. 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 central Safety-nucleus is based on a TMR (Triple Modular Redundancy) configuration of computers implementing a two out of three voting scheme, with automatic exclusion of the unit in disagreement with the other two.

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 control system is represented by a set communicating processes, modelling logical and physical entities. The control of the entities is realised by operations which act on variables. Often variables represents signals whose domain of values is very limited or a limited number of values are of interest. The specification and verification of the system is reported in [2].

The translation from the semi-formal to the formal specification was straightforward as shown in figure 13 and figure 14.

Each operation in the Ansaldo semi-formal specification can be described in three main parts:

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

```
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

6.1.1 Reduction of the number of states

The use of the abstraction techniques presented above for testing signal values, before combining the replicas of the TMR configuration by means of the tools of the Jack environment, have produced a model of the behaviour of the system composed by one replica of about one million of states. The use of static parameters allowed a reduction in the number of states of this global LTS from about one million states to 77294 states.

6.1.2 Safety requirements verification

A typical safety requirement for an interlocking system is that if a train is entering a track containing a level crossing, if the *proceed signal* is sent to the train at the

```

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

beginning of the track then the position of the level crossing is *closed*. This property can be expressed more precisely as a proposition on the model of the behaviour of the system as follows: in any state of the model if the position of the level crossing is not equal to *closed*, then there is not an execution in which the *proceed signal* is sent until the position of the level crossing is equal to *closed*. Hence, this property can be classified as a *fail-safe* property.

This expression can be formalised as a formula in the ACTL logic as follows. Let `raise_shunt_sign` be the action corresponding to the *proceed signal* and let `on_pos`, `off_pos` and `undefined_pos` be the different positions that the level crossing can assume:

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

6.2 Inter-consistency mechanism

The GUARDS project has produced a generic architecture for safety critical systems [42] designed to be instantiated to support different critical applications. Model checking techniques have been used in the project to validate the Inter-Consistency mechanism which is the basis of the ad hoc defined fault tolerant mechanisms.

Interactive consistency focuses on the problem of reaching agreement among multiple processors in presence of faults (also known as the "Byzantine Generals problem" [33]). The principal difficulty to be overcome in achieving interactive consistency is the possibility of conflicting values sent by faulty processors.

The Inter-consistency mechanism uses the ZA Byzantine Agreement algorithm described in [25]. According to the GUARDS architecture, the Inter-Consistency mechanism must guarantee consistency among three or four processors. The 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. The assumption of message authentication requires that faulty processors do not make undetectable modifications to messages as they are relayed from one processor to another. The mechanism is a composition of transmitter and receiver protocols: for example, in 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 algorithm is modelled as a network of four communicating processes, each modelling one of the four nodes. Moreover, the algorithm has a phased structure:

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

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

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

each of the previous processes is described by a network communicating processes modelling the different phases of the algorithm and the local variables. We refer the reader to [3] for the complete specification and verification work.

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 upon receiving a message from S (or detecting an omission fault), saves the message into the variable named **m1p**. Then it is ready to execute **phase 3** of protocol, and signals this by the **!startphase3** action, on which all the other nodes have to synchronize.

6.2.1 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 constraint the occurrences of faults.

The table clearly shows:

- the fact that the size of the state space is largely below the fourth power of the size of the state space of a single node confirms the observations we have enunciated previously;

- the increase of the state space with the generality of the fault assumptions, evident in the last two rows.

6.2.2 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.

The formalisation of these properties as ACTL formulae is:

Agreement:

for any execution of the processes,

the nodes eventually agree on the value 1 (actions `!vp_ofp_eqto_1`, `!vp_ofq_eqto_1`, `!vp_ofr_eqto_1`, `!vp_ofs_eqto_1`) or the nodes eventually agree on the value 0 (actions `!vp_ofp_eqto_0`, `!vp_ofq_eqto_0`, `!vp_ofr_eqto_0`, `!vp_ofs_eqto_0`).

Validity:

if in any state of the model, it is true that the internal value of the node P is equal to 1 (action `!psend_vp_1`) or 0 (action `!psend_vp_0`), then for any execution of the processes, starting from such a state, the nodes eventually agree on such a value.

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

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

We applied the model checker tool to prove the invariance of required properties under given fault assumptions. As expected, we found that in the case of a violation of the assumption on authentication, even a single faulty node is not tolerated.

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 case studies from industries and confirm that key-point in the industrial acceptance of model checking are

- the using 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).
- the existence of automatic verification tools.

State explosion represents the main problem to the application of model checking for handling large industrial systems in many fields. 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.

Acknowledgments

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] Bowen J, Stavridou V. 1992. Safety-critical systems, formal methods and standards. *PRG-TR-5-92*, Oxford University Computing Laboratory.
- [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 conferece 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] Cimatti A, Giunchiglia F, Mongardi G, Romano D, Torielli F, Traverso R. 1998. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, **10**(4).

- [15] 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.
- [16] Cristian, F. 1985. A rigorous approach to fault tolerant programming. *IEEE Transaction on Software Engineering*, **11**(1): 23-31.
- [17] Damm W, Klose J. 2001. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, **19**(2).
- [18] 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.
- [19] 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.
- [20] Eisner C. 1999. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. *Proceedings 10th IFIP WG 10.5 Advanced Research Working Conference*, CHARME '99, Germany.
- [21] Emerson EA, Sistla AP. 1993. Symmetry and model checking. *Proceedings Computer Aided Verification'93*, Lecture Notes in Computer Science **697**, Springer, Berlin.
- [22] Fantechi A, Gnesi S, Pugliese R, Tronci E. 1998. A symbolic model checker for ACTL. *Proceedings FM-Trends'98: International Workshop on Current Trends in Applied Formal Methods*, Lecture Notes in Computer Science **1641**, Germany.
- [23] 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.
- [24] Fokkink WJ, Hollingshead PR. 1998. Verification of interlockings: from control tables to ladder logic diagrams. J.F. Groote, S.P. Luttik and J.J. van Wamel, eds. *Proceedings 3rd Workshop on Formal Methods for Industrial Critical Systems*, FMICS'98, Amsterdam, Stichting Mathematisch Centrum : 171-185.
- [25] 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.
- [26] 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.
- [27] Hennessy M, Milner R. 1985. Algebraic laws for nondeterminism and concurrency. *ACM*, **32**(1): 137-161.
- [28] Holzmann GJ, Peled D. 1994. An improvement in formal verification. *Proceedings FORTE 1994 Conference*, Bern, Switzerland.
- [29] 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.

- [30] 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.
- [31] Kozen D. 1983. Results on the propositional μ -calculus. *Theoretical Computer Science*, **27** : 333-354.
- [32] Krishnan P. 1994 A semantic characterisation for faults in replicated systems. *Theoretical Computer Science* **128**: 159-177.
- [33] Lamport L, Shostak R, Pease M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982; **4**(3): 382-401.
- [34] 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.
- [35] Laprie JC (Ed.). 1992. *Dependability: basic concepts and terminology*. Dependable Computing and Fault-Tolerant Systems, **5**, Springer-Verlag.
- [36] Liu Z, Joseph M. 1992. Transformation of programs for fault tolerance. *Formal Aspects of Computing*, **4**: 442-469.
- [37] Manna Z, Pnueli A. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer-Verlag.
- [38] Milner R. 1989. *Communication and concurrency*. Prentice-Hall International, Englewood Cliffs.
- [39] 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.
- [40] Peled D, Joseph M. 1994. A compositional framework for fault tolerance by specification transformation. *Theoretical Computer Science* **128**: 99-125.
- [41] Peleska J. (1990) Design and verification of fault tolerant systems with CSP. *Distributed Computing*, **5** (2): 95-106.
- [42] 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.
- [43] 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 Edinburg.
- [44] 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.
- [45] Schepers H, Hooman J. 1994. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science* **128**: 127-157.
- [46] Schneider FB. 1990. Implementing fault tolerant services using the state machine approach: a Tutorial. *ACM Computing Surveys* **22** (4): 299-319.