

Model Checking Embedded 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

In this paper we show how embedded fault tolerant systems can be analysed by model checking formal verification technique. We first review how formal methods have so far been applied in the development of fault tolerant systems, then we discuss the critical points for the success of the introduction of model checking technique in such a field, finally we present a modelling approach suitable for model checking fault tolerant systems under different fault scenarios. 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

In the development of embedded computer-controlled systems, a combination of the methods of fault prevention, fault tolerance, fault removal and fault forecasting are used in order to achieve high dependability. In particular, it is commonly agreed that a viable means to reduce the failure rate of a program is the use of fault avoidance in the form of formal methods in conjunction with other techniques [1].

The application of formal methods in the rigorous definition and analysis of the functionality and the behaviour of a system, promises the ability of showing that the system is correct.

Given such a promise, that is already out since several years, it is astonishing to see how little formal methods are actually used in the safety critical system industry, though the use of formal methods is increasingly required by the international standards and guidelines for the development of safety critical computer-controlled systems.

Industrial acceptance of formal methods is strictly related to the investment needed to introduce them, to the maturity of tool support available, and to the easiness of use of formal methods and tools.

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.

Several approaches to the application of formal methods in the development process have been proposed, differing for the degree of involvement of the method within it. Starting from rigorous specifications, formal methods can be used for the derivation of test cases, or as a validation technique aimed to prove that the specification satisfies the requirements, or as an auxiliary technique in the automated generation of code.

Formal verification methods based on model checking work on a finite state representation of the behaviour of the system. Verification is usually carried out by checking the satisfiability of some desired properties formalized as logical formulae over the model of the system by model checking algorithms. As an example, safety requirements may be expressed as temporal logic formulae and may be checked on the model of the system. This provides a direct automatic verification method of system properties; unfortunately, this approach suffers of the so called "State Space Explosion" problem: 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. Hence, traditional model checking techniques [2], have shown themselves not to be powerful enough to cope with many "real" systems, when their models are larger than 100000 states.

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 [3]. In SMV the transition relations are represented implicitly by means of Boolean formulae and are implemented by means of Binary Decision Diagrams (BDDs, [4]). 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.

Embedded computer-controlled systems often include fault tolerance techniques. Fault tolerance is the property of a system to provide, by redundancy, a service complying with the specification in spite of faults occurred or occurring [5]. A failure of a system occurs when the behaviour of the system first deviates from that required by the specification. The formalization of fault tolerance includes modelling faults and failures, as well as fault tolerance schemes.

In this paper we present an approach suitable for formally studying the behaviour of a fault tolerant system under different fault scenarios. The verification of fault tolerance properties is based on model checking. We discuss model checking and its application to fault tolerant systems, and we show how particular characteristics of this class of systems allow the state explosion problem to be tackled.

The paper is organized as follows. Section 2 reviews model checking application to computer-controlled systems. In section 3 we present our technique for modelling embedded fault tolerant systems. Section 4 review 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 two case studies of fault tolerant systems and their properties. In the final Section 7 we give some concluding remarks about future research.

2 Model Checking

Model checking is a formal verification technique that, literally, means “Checking (properties on, correctness of) a Model” of a system [2]. Actually, we can find two interpretations of the term Model Checking:

- The first, canonical, interpretation, widely adopted in the formal verification community, denotes as model checking any *algorithmic* and *exhaustive* verification on the model of a system: this includes temporal logic model checking as well as reachability analysis, equivalence or preorder checking and so on: in any case, the whole state space is taken into account. If some optimization which does not actually visit every state is used, it is because some formal reasoning can be applied to show that what is proved on the reached states guarantees the result of the verification on the whole state space.
- The second interpretation is more relaxed, and includes a *non exhaustive* verification techniques such as simulation/animation of the model, in which only some possible executions of the system are analyzed. Some commercial tools, such as the ones working on SDL[6], offer advanced simulation techniques that approximate exhaustive search and provide, together with provisions to cut the state space in depth or width, some techniques to calculate a structural coverage measure of the performed verification with respect to the whole model. These are very similar to the coverage measures used in software testing. As for testing, a 100% coverage measure does not automatically mean that the verification made is exhaustive; nevertheless the sophisticated techniques used for the generation of the state space (which have been mostly inherited by the research on model checking) may allow exhaustiveness to be reached in many cases.

Some of these tools are currently used extensively in the telecommunication industry, and are gaining acceptance also in the safety critical systems field.

We advocate however the use of the first interpretation that distinguishes “complete” formal verification (that algorithmically gives an yes/no result) from “partial” verification: a complete verification is what is primarily expected by an embedded fault tolerant system validation technique.

Using model checking, we can study whether a property formalised as a logic formula holds for a system model. Interesting properties of fault tolerant systems are:

1. *Correctness*. The system delivers a correct service.
2. *Fault tolerance*. Despite of 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 have to be studied with respect to specific classes of faults and in presence of given fault occurrences, that is, under well-defined *fault assumptions*. All the system properties above can be formally specified as logic formulae in a temporal logic, whose operators permit explicit quantification over all possible futures.

3 Modelling Fault tolerant Systems

In this section we present how fault tolerant systems can be specified in such a way that the specification can be analysed by model checking technique.

We use the following concepts and terminology.

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

Definition 2 (failure mode) Failure mode denotes the specification of the behaviour of system after the occurrence of a fault.

Definition 3 (failing system) Failing system denotes the complete specification of the system considering the possible occurrence of faults. After the occurrence of a fault, the failure mode is exhibited.

Definition 4 (fault tolerant system) Fault tolerant system denotes the specification of the behaviour of the system after the application of a fault tolerance technique.

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

Our approach to the formalization of fault tolerant system 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. We assume faults to be 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 in our approach to specify a system: the CCS/Meije process algebra or an equivalent graphical notation.

CCS/Meije. The syntax of CCS/Meije [7] 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 syntax relies on the following assumptions:

1. Act , ranged over by α is a set of action names. Such names represent emitted signals if they are prefixed by "!" or received ones if they are prefixed by "?";
2. τ denotes a special action not belonging to Act . Action τ represents an internal action;
3. $Act_\tau = Act \cup \{\tau\}$, ranged over by a, b , denotes the full set of actions that a process can perform;
4. X , ranged over by X , is the set of term variables.

The following grammar generates all regular terms, ranged over by R , and all network terms, ranged over by P :

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

$$P ::= R \mid P \parallel P \mid P \setminus a \mid P[a/b]$$

where $[\dots]$ denotes an optional and repeatable part of the syntax.

Informally,

- an inactive process is specified by the **stop** operator;
- the action prefix operator ($a : R$) specifies the execution of actions in sequence;
- the nondeterministic choice operator ($R + R$) indicates that a process can choose between the behaviour of several processes;
- Parallel composition of two processes ($P \parallel P$) corresponds to the interleaved execution of the two processes;
- The restriction operator ($P \setminus a$) indicates that an action can only occur within a synchronization. This operator is used to specify processes which communicate (synchronise on actions). The restriction operator transforms the couple of actions executed together into the internal action τ ;
- The relabeling operator ($P[a/b]$) transforms an action into another action.

The semantics of CCS/Meije terms is given operationally over Labelled Transition Systems (LTSs):

Definition 6 *An LTS is a 4-tuple $\mathcal{A} = (X, x^0, Act_\tau, \rightarrow)$, where: X is a finite set of states; x^0 is the initial state; Act is a finite set of observable actions; $\rightarrow \subseteq X \times Act_\tau \times X$ is the transition relation. In particular, $x \xrightarrow{a} x'$ denotes the transition from the state x to the state x' by executing action a .*

For the complete operational description, we refer the reader to [7].

Example – Figure 1 reports the CCS/Meije specification of a simple system that maintains the position of a level crossing gate (**gate_contr_p** in the following). The system is a recursive process with three different states: **UNDEFINED_P**, **ON_P** and **OFF_P**. The initial state of the process is **UNDEFINED_P** and, in each state, the process is able to send a signal indicating its current state (**on_p!** in the state **ON_P**, similarly for the state **OFF_P** and **UNDEFINED_P**). It can change state by receiving a signal **s_on_p!** (set state on) or **s_off_p** (set state off). Moreover, the initial state **UNDEFINED_P** is not reachable by the other states.

The specification of the **gate_contr_p** system together with the operations of opening and closure of the level crossing (**open_op** and **close_op**, respectively) is reported in figure 2. In this case the parallel composition operator is used. The specification is named **net**.

While the **open_op** and **close_op** processes are independent each other, these processes must synchronise with the **gate_contr_p** when checking the level crossing position or sending a signal to change the level crossing state (actions: **on_p**, **off_p**, **undefined_p**, **s_off_p** and **s_on_p**). \square

Graphical notation. The graphical notation we use, defined for the ATG tool [8], expresses a sequential process by drawing the LTS representing its behaviour

```

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 1: The `gate_contr.p` specification

```

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

```

Figure 2: The `system_net` specification

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.

The graphical formalism allows the synchronisations between processes to be observed by setting a label on the edge connecting the corresponding ports. Moreover, a multiway synchronisation operator is also available. In this case we can model the situation in which more than two processes must synchronise on a given action.

Example – Figures 3 and 4 report the graphical specification of the the `gate_contr.p` and the `net` systems, respectively.

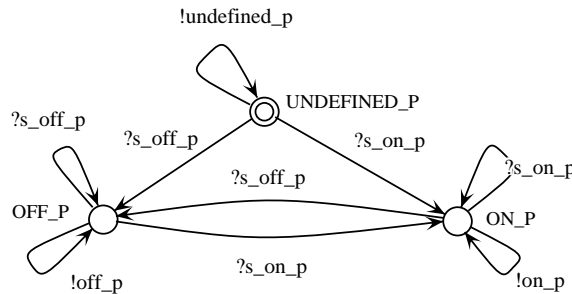


Figure 3: A sequential process

By setting the label `OFF_COMMAND` on the edge linking ports `!s_off_p` and `?s_off_p`

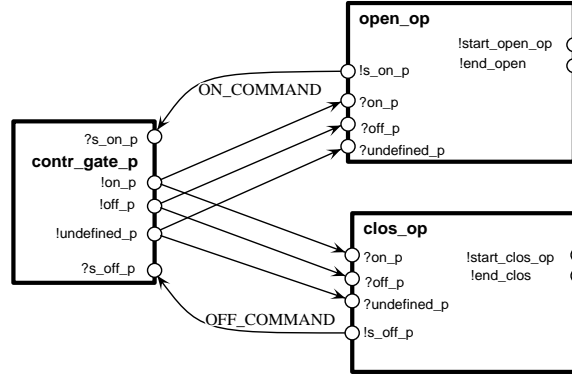


Figure 4: A network of processes

in figure, each time processes synchronise by executing `!s_off_p` and `?s_off_p`, we observe `OFF_COMMAND`. Similarly for the edge linking the ports `!s_on_p` and `?s_on_p`. \square

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 failing 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 subsystem, then the failure mode of the subsystem is exhibited, otherwise, the subsystem goes on with its behaviour.

We can assume for generality that the failure mode of the process depends on the point at which a fault occurs during the execution of the system. In most cases, such high granularity of 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 may produce a coarser granularity. Some example in this direction are:

1. confining faults to specific subsystems;
2. choose specific points in the execution of the subsystems at which a fault may occur, realizing 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. For example it stops.

Example – Figure 5 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 faults can occur at every non-faulty state.

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 loose the current correct state, by showing the value `undefined` until the reception of a signal setting the position of the level crossing. We assume the general condition that a fault may occur at any time. An output edge labelled by

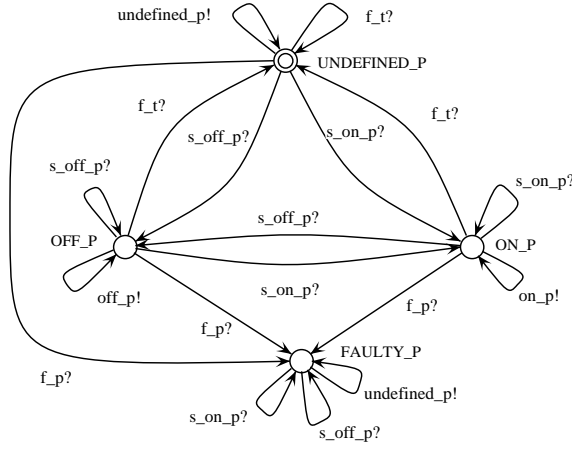


Figure 5: The failing system `contr_gate_p`

$?f_p$ and an output edge labelled $?f_t$ exists starting from each state of the entity except the `FAULTY_P` one. \square

3.3 Introducing fault tolerance

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. 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). Replicas may be composed together with some extra standard components added by the fault tolerance technique (for example, a Voter) for hiding the effects of the occurrence of faults by fault detection or correction.

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.

If n is the number of replicas used by the fault tolerance technique, \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.

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 processes are used with appropriate renaming of the actions.

We note that M may be empty and that often the fault masking techniques (for example a Triple Modular Redundancy - TMR) could be expressed as a context which takes only one argument, the failing system, and generates the required number of instances of the argument with appropriate renaming of the actions. The

distinction among the arguments is more general, since it allows us to specify in the same way also fault tolerance techniques based on design diversity, in which instead of replicas, variants are used, each of which corresponds to a particular specification of the system.

Finally, if error detection is applied, 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.

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. Similarly to the *fault assumption process*, a process named *recovery assumption process* can be included in the specification to express constraints on the recovery.

Example – Figure 6 reports the specification of the duplicated version with comparison of the system `contr_gate_p` under the fault assumption that at most one permanent fault may occur in one of the replicas. \square

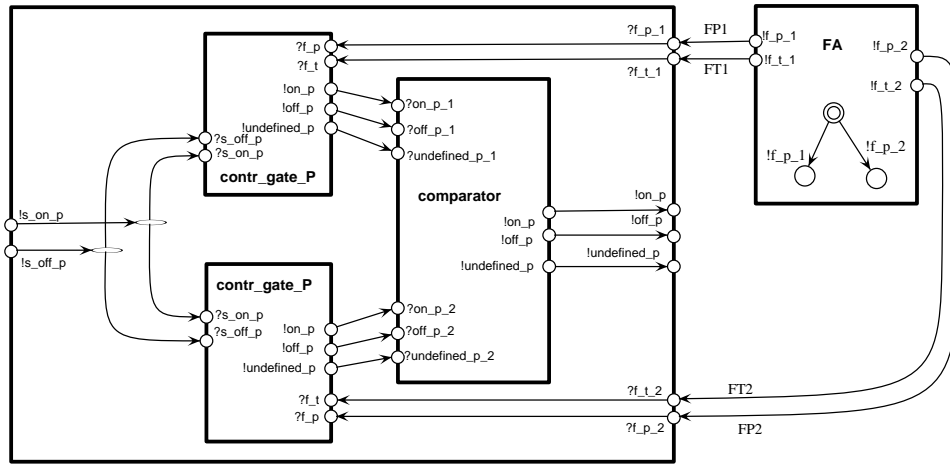


Figure 6: Fault tolerant system

3.5 Generation of the global system model

As we will see in the next section the automatic verification of properties of the fault tolerant system requires the generation of the state machine representing its overall behaviour (*global LTS*). This generation can be automatically performed by means of tools based on the standard operational semantics rules of process algebras [9]. In particular, we refer here to the tools available inside the JACK [10] environment. Every state of the global LTS represents the combined current states of the subsystems components. It is at this stage that the so called *state explosion problem*, which we will discuss later, occurs.

Example – A part of the global LTS for the fault tolerant system in figure 6 is reported in figure 7, where INITIAL is the initial state and states with a double cir-

cle (except INITIAL) represent unexplored states. The global LTS has 100 states. \square

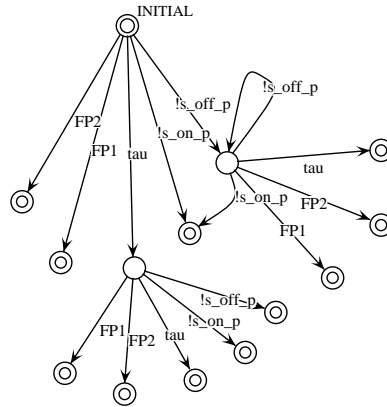


Figure 7: A part of the global LTS of the system in figure 6

4 Properties for embedded fault tolerant systems

Temporal logic can be used to express properties of a system [11]. We use a temporal logic in agreement with the selected specification formalism to formalise system's properties.

4.1 The logic

The particular temporal logic we use is called ACTL (Action-based Computation Tree Logic) [12], which is an action-based version of the branching time temporal logic CTL [2]. 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 [13].

The formulae of ACTL are *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 .

The informal semantics of the used ACTL operators is shown in Table 1 (the formal one is given in [12]). In the table, α is an action belonging to the set Act of actions executable by the system, \sim is the *negation* operator, E and A are the existential and universal path quantifiers, while U is the *until* operators.

4.2 Formalising properties

System safety properties are usually invariant, so the formulae we want to check are of the form $AG\phi$, which means that ϕ should hold in every state. We can formalize the typical properties we wish to prove of a fault tolerant system along the following schemes:

- *Correctness*

$AG \text{ FCorr},$ where FCorr expresses a correctness condition

Action formulae

$\chi ::= true$	any observable action
α	the observable action α
$\sim \chi$	any observable action different from χ
$\chi \mid \chi'$	either χ or χ'

State formulae

$\phi ::= true$	any behaviour is possible
$\sim \phi$	ϕ is impossible
$\phi \ \& \ \phi'$	ϕ and ϕ'
$E\gamma$	there exists an execution in which γ
$A\gamma$	for every execution γ
$< \alpha > \phi$	there exists a next state reachable with α , in which ϕ
$[\alpha]\phi$	for all next states reachable with α , ϕ holds

Path formulae

$\gamma ::= G\phi$	at any time ϕ
$F\phi$	there is a time in which ϕ
$[\phi\{\chi\}U\{\chi'\}\phi']$	at any time χ is performed and also ϕ , until χ' is performed and then ϕ'

Table 1: Syntax and informal semantics of the used ACTL operators.

- *Fault tolerance*

$AG \ [fault] \ FCorr$

- *Fail-stop*

$AG \ [fault] \ FTerm,$ where $FTerm$ expresses the termination of the system

- *Fail-silence*

$AG \ [fault] \ FCorrOmiss,$ where $FCorrOmiss$ expresses the correctness,
apart from omission failures

- *Fail-safe*

$AG \ [fault] \ \sim FUnsafe,$ where $FUnsafe$ expresses an unsafe behaviour

Clearly, the formulae $Fcorr$, $Fterm$, $FcorrOmiss$ and $Funsafe$ are strictly dependent on the functionality of the system.

We have then to note that, although safety properties are of main concerns, the logic offers the possibility of checking other desirable properties as well. These can be in general expressed as AF or EF formulae.

Example – Correctness property. The `contr_gate_p` system whose model is shown in figure 3 always shows a state equal to the last received set state to signal. For the `?s_on_p` signal, this is expressed by the ACTL formula:

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

The formula states that the system, after being in the state `on` cannot be into the state `off` until a set to off signal has been received.

Fail-safe property. The failing `contr_gate_p` system whose model is shown in figure 5 satisfies the property that a state equal to the last received set state to signal or the state undefined is shown. For the `?s_on_p` signal, this is expressed by the ACTL formula:

```
AG[?s_on_p]A[~!off_p {true} U {?s_off_p} true]
```

Fault tolerance property. The fault tolerant configuration in figure 6 whose model is shown in figure 7 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. The property can be written as: after a fault, the system always shows a state equal to the last received set state to signal. For the `?s_on_p` signal, this is expressed by the ACTL formula:

```
AG[FP1]AG[?s_on_p]A[!on_p {true} U {?s_off_p} true] &
AG[FP2]AG[?s_on_p]A[!on_p {true} U {?s_off_p} true]
```

□

4.3 Properties verification

The model checker accepts a finite state machine (LTS) and an ACTL formula [14]. 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.

Example – Consider the failing `contr_gate_p` system and the property stating that every execution shows the value `on` at least once:

```
AF<!on_p> true
```

The formula is obviously *false* on the LTS in figure 5, for every path which does not include the `?s_on_p` action.

We obtain the following trace from the counterexample facility:

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

□

5 Dealing with state space explosion problem

The main difficulty in using in practice formal verification methods is due to the limits imposed by the size problem, that even challenges more advanced model checking tools. The use of techniques such as decomposition and abstraction, to overcome the state space explosion problem at the specification level, are only partially successful, and require a great deal of expertise which industries often do not have. Thus current techniques have failed to arrive at the level of usability required by the industrial applications.

A solution could be the development of domain-specific optimization of model checking algorithms. 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 [15] 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.

We show here that some characteristics of embedded fault tolerant systems, such as redundancy and static configuration parameters provide opportunities to limit a priori the state explosion problem, even if adopting traditional model checking algorithms.

5.1 Redundancy

5.1.1 Phased structure of fault tolerant systems and algorithms

The formal modelling of a fault tolerant system can be often structured as a network of replicas, each divided in phases of useful work; at the end of each phase the replicas synchronize to maintain consistency through exchange of messages. Indeed, a system employing redundancy is composed by a number of identical modules which compute the same results. At the architecture level such modules are often, in today's embedded systems, 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 redundant system is therefore a distributed system that uses specialized interaction protocols. Usually such protocols have to ensure some notion of consistency even in presence of faults, and trigger appropriate corrective actions. The formal verification of such protocols is therefore an important step in the establishment of the overall correctness and safety of the system even in presence of faults.

A common structure of such a system can be represented (in the case of duplication redundancy) as shown in Figure 8, as a network of automata; each LTS synchronizes with the other ones at the end of each phase. In general, more than a single synchronization action is involved at the end of a phase. Here we abstract such a complex synchronization protocol with only a single action, without affecting the validity of the following discussion.

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, leading to a single global state from which the next phase begins (see Figure 9, 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: $S_1^n + S_2^n + \dots + S_m^n$, which is significantly lower.

5.1.2 System replication

The regular structure of a redundant system may be exploited to contain state explosion with the help of established techniques, such as *symmetries* and *reduction*

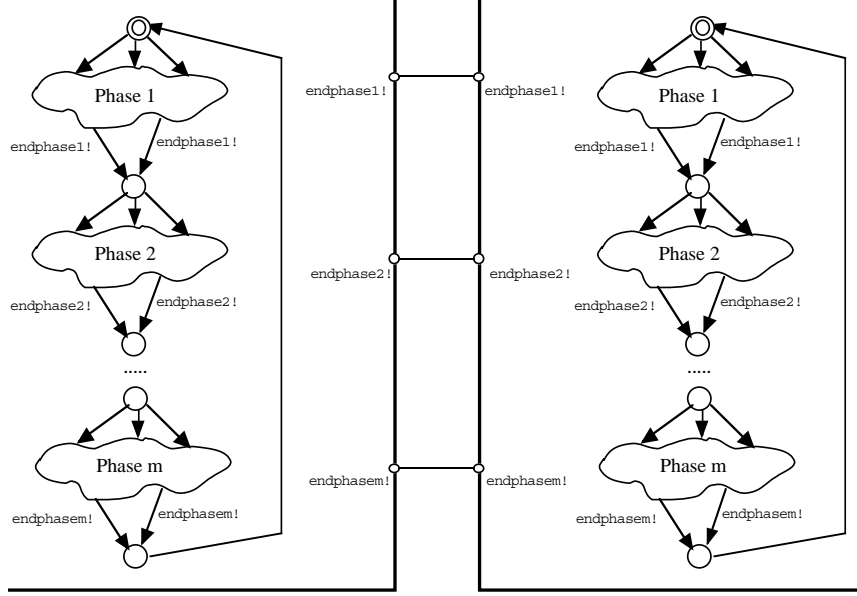


Figure 8: The phased structure

preorders. Using symmetries, as proposed by Emerson e. g. in [16], the number of states is reduced by identifying those states which coincide up to a permutation of the system components. Reduction preorders [17] 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, replicas can be considered as largely independent one from another. We can therefore avoid to generate the complete interleaving of the replicas in the generation of the model. We can select only those particular executions in which all the transition contributed to the global LTS by the first replica are executed first, then the second replica starts, and so on.

The selection has however to take into account the interactions between the replicas. In the case of reduction preorder applied to phase structured systems, for example, the corresponding phases of the replicas are completely independent, since the replicas interact only at the synchronization points. Hence, the global state space of a phase i of the global LTS can be reduced to be of the order of $n * S_i$, and therefore the global state space of the overall algorithm can be reduced to $n * S$.

5.1.3 Fault and failure modelling

Modelling a fault tolerant system means also to include in the model a description of the behaviour of the system in case faults occur, in order to be able to prove fault tolerance properties.

The modelling of fault occurrences and of relative corrective actions is a major source of complexity; moreover, they tend to break independency and similarity of replicas.

However, it is important to notice that it is often not interesting to prove the safety critical properties in any general fault scenario, but it is enough to consider only some restricted fault scenarios. Typically, in a redundant system, assumptions are made on the maximum number of admitted faulty replicas. An accurate definition

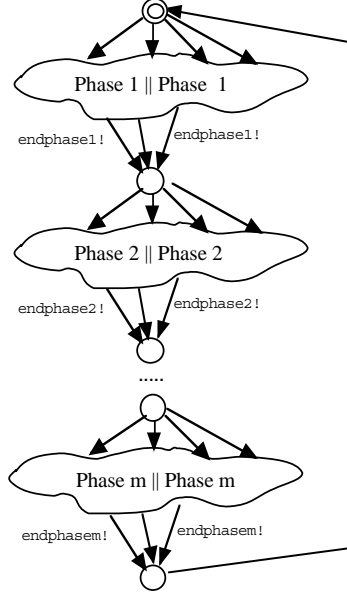


Figure 9: The phased structure

of the fault assumptions, which is usually an important part of the validation of safety critical systems, 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 pahses, 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.

The combined use of all the techniques we have shown can reduce dramatically the expected state explosion of a redundant system.

5.2 Static configuration parameters

Generally, the original semi-formal specification of an embedded system takes into account some *attributes* that can be considered as static configuration parameters and describe the particular type entity that is controlled. As an example, a typical attribute for a level crossing entity says whether it is on the mainline or on a parking area. The semi-formal specification considers these attributes as if they were variables. This would contribute unnecessarily to the growth of the number of states of the model. Therefore, in the development of the formal specification we can take the configurations each at a time. A safety property is satisfied if the the property is verified in all possible configurations.

5.3 Testing signal values

The following technique can be applied for the case in which an entity 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. If the safety properties do not involve actions related to the tested signals, the actions corresponding to a

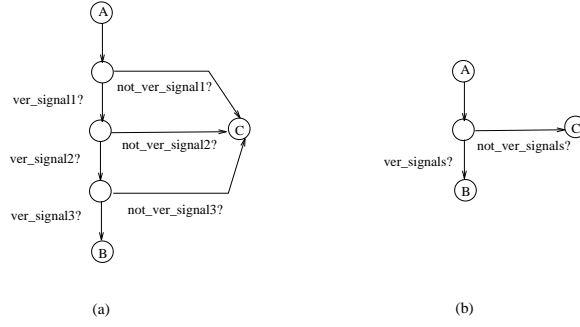


Figure 10: (a) A fragment of a general operation. (b) The fragment after the reduction of the sequence of tests.

sequence of tests can be modelled as a non deterministic choice between the success and the failure of the tests.

Let us consider, for example, the case in which a system executes sequentially a test on three different signals represented by the LTS in figure 10 (a). The failure of any test leads to state C, while the success of all the tests leads to state B. We can rewrite the specification as shown in figure 10 (b), in which the sequential tests are substituted by a simple abstract test which may fail, leading to state C; or it may be executed with success leading to state B.

6 Case studies

We have used our analysis technique to specify and verify 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 [18]. 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) [19]. Both studies show that:

- the application of model checking formal verification methodology is feasible and well accepted in the industrial context of embedded fault tolerant systems;
- the formalization process strictly depends on the application tipology. 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 embedded 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;
- the use finite state machine as specification language has the advantage of ensuring the adherence of the produced formal specification to the original semi-formal one.

6.1 Railway Interlocking System

The first case study is a part of a railway signaling interlocking control system developed by Ansaldo Trasporti. 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 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 [18].

The translation from the semi-formal to the formal specification was straightforward as shown in figure 11 and figure 12.

Each operation in the Ansaldo semi-formal specification can be described in three main parts: some conditions on variables must be satisfied before continuing the operation (“VERIFY THAT” part). 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 11: Semi-formal specification

```
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 12: Formal specification

6.1.1 Reduction of the number of states

Using the techniques presented above for testing signal values and combining the replicas of the TMR configuration, we have obtained a model of the behaviour of the system composed by one replica of one million of states. The 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 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*.

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 ( [~!off_pos]
      (~E[true {~ ?s_off_pos} U (<!raise_shunt_sign> true)]) ) )
```

6.2 Inter-consistency mechanism

The GUARDS project has produced a generic architecture for safety critical systems [19] 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" [20]). 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 [21]. 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 2, 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: 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 [22] 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:

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 2: The ZA algorithm of transmitter node P

Table 3: 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

```

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 3 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;
- the increase of the state space with the generality of the fault assumptions.

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:

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

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 Discussion and conclusion

In the literature on the formalisation of fault tolerant systems, several works are based on process algebras and equivalence relations or preorders to verify fault tolerant system designs. The major advantage of such approaches is related to the existence of automatic verification tools. A short survey of such approaches can be found in [23].

The application of model checking in the field of fault tolerant systems is quite new. Model checking of properties expressed in modal μ -calculus is applied in [24] to analyse fault-handling mechanisms. The mechanisms are usually 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. A fault extends the behaviour of a system by allowing the fault to occur in any state.

In our model checking approach we have preferred to stick to traditional process algebras, in order to be able to exploit the powerful verification capabilities offered by existing verification environments.

We believe that model checking can play a major role in the validation of embedded fault tolerant systems.

Model checkers are ready to be introduced in the industrial development process, aside traditional development techniques. Still state explosion represents the main problem for handling large industrial systems in many fields, but the current research efforts in search of powerful algorithms are promising. In particular, we have shown in this paper how the analysis of typical characteristics of fault tolerant systems can be exploited to tackle state explosion problems.

These results are not depending on the particular formalism chosen to model fault tolerant systems and can be applied also if diverse redundancy is employed.

The ability of using specific techniques to deal with state explosion problem, like the ones we have shown for redundant systems, provides the opportunity of defining a specific formal verification process for a given application field, thus decreasing the need of general verification expertise in industries.

A key-point in the industrial acceptance of model checking is that it relies on models which are essentially (some variants of) finite-state machines, which are commonly used in many industrial activities, especially in the safety critical systems area.

We can observe that railway industries have assessed the introduction of formal verification by model checking in their development processes before other safety critical industries, and with a greater success. We can refer to [18, 25, 26] for some notable examples, but there have been many other published and unpublished work in this direction.

In railway signalling industry, the safety critical part of a control system is the so called interlocking logic, whose main aim is to guarantee, through the typical use of a transition to a safe state (that is, by stopping the trains), that the system does not enter a critical state. Interlocking logic is usually amenable to be formalized through the use of state machines, or of first-order logic predicates, and this does not usually require a large investment in people. Moreover, safety properties to be proved on the system depend on the combination of values of discrete variables, and this makes state explosion problems easier to be attacked.

In medical systems, as well as automotive, avionic or space systems, on the other hand, the safety characteristics of the system are often controlled by sophisticated, numerical algorithms; therefore, the safety properties depend on the combination of values of continuous variables with often more stringent real time requirements. Though specific model checking technologies have been developed to cope with such systems (such as timed and hybrid model checking), continuous variables and time add such a complexity to the state space, that current model checking technologies appear not mature enough for an heavy industrial usage. This difference in the nature of the controlled process is the responsible for the slower acceptance of formal verification techniques in these industries. The same delay has been observed, within the railway industries itself, about the application of model checking to the verification of on-board equipments, since they introduce dependency from real quantities (speed, time, etc...) and hence increasing the complexity and the size of the state space [27].

References

- [1] Bowen J, Stavridou V. Safety-critical systems, formal methods and standards. *PRG-TR-5-92*, Oxford University Computing Laboratory, 1992.
- [2] Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 1986; **8**(2): 244-263.
- [3] Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 1992; **98**(2): 142-170.
- [4] Bryant RE. Graph based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 1986; **C-35**(8).

- [5] Laprie JC (Ed.). *Dependability: basic concepts and terminology*. Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer-Verlag, 1992.
- [6] Z100: Specification and description language SDL. ITU-T, June 1994.
- [7] Austry D, Boudol G. Algebre de processus at synchronisation. *Theoretical Computer Science*, 1984; 1(30).
- [8] Roy V, De Simone R. AUTO and Autograph. *Proceedings of the Workshop on Computer Aided Verification*, Lecture Notes in Computer Science **531**, 1990, 65-75.
- [9] Milner R. *Communication and concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
- [10] Bouali A, Gnesi S, Larosa S. The integration project for the JACK environment. *Bulletin of the EATCS*, 1994; **54**: 207-223.
- [11] Manna Z, Pnueli A. *The temporal logic of reactive and concurrent systems - specification*. Springer-Verlag, 1992.
- [12] De Nicola R, Vaandrager FW. Actions versus state based logics for transition systems. *Proceedings Ecole de Printemps on Semantics of Concurrency*, Lecture Notes in Computer Science **469**, Springer, Berlin, 1990, 407-419.
- [13] Hennessy M, Milner R. Algebraic laws for nondeterminism and concurrency. *ACM*, 1985; **32**(1): 137-161.
- [14] Fantechi A, Gnesi S, Pugliese R, Tronci E. 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, 1998.
- [15] Eisner C. 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, 1999.
- [16] Emerson EA, Sistla AP. Symmetry and model checking. *Proceedings Computer Aided Verification'93*, Lecture Notes in Computer Science **697**, Springer, Berlin, 1993.
- [17] Holzmann GJ, Peled D. An improvement in formal verification. *Proceedings FORTE 1994 Conference*, Bern, Switzerland, 1994.
- [18] Bernardeschi C, Fantechi A, Gnesi S, Larosa S, Mongardi G, Romano D. A formal verification environment for railway signaling system design. *Formal Methods in System Design*, 1998; **12**: 139-161.
- [19] Powell D, Arlat J, Beus-Dukic L, Bondavalli A, Coppola P, Fantechi A, Jenn E, Rabjac C, Wellings A, GUARDS: a generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 1999; **10**(6), 580-599.
- [20] Lamport L, Shostak R, Pease M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982; **4**(3): 382-401.
- [21] Gong L, Lincoln P, Rushby J. 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, 1995.

- [22] Bernardeschi C, Fantechi A, Gnesi S. Formal validation of fault tolerance mechanisms inside GUARDS. *Proceedings SAFECOMP'99*, Toulouse, Lecture Notes in Computer Science **1698**, 1999.
- [23] Bernardeschi C, Fantechi A, Simoncini L. Formally verifying fault tolerant system designs. *The Computer Journal*, **43**(3), 2000.
- [24] Bruns G, Sutherland I. Model checking and fault tolerance. *Proceedings 6-th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science **1349**, Sydney, Australia, 1997, 45-59.
- [25] Cimatti A, Giunchiglia F, Mongardi G, Romano D, Torielli F, Traverso R. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, 1998; **10**(4).
- [26] Fokkink WJ, Hollingshead PR. 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, 1998, 171-185.
- [27] Damm W, Klose J. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, 2001; **19**(2).