Application of Formal Methods for Validating an Interaction Policy

A. Fantechi [1], S. Gnesi [2], L. Semini [3]

1. Dipartimento di Sistemi e Informatica, Università di Firenze
e–mail: fantechi@dsi.unifi.it

2. Istituto di Elaborazione dell'Informazione, C.N.R., Pisa
e–mail: gnesi@iei.pi.cnr.it

3. Dipartimento di Informatica Università di Pisa
e–mail: semini@di.unipi.it

Abstract

Formal methods are increasingly being used to validate the design of software and hardware components of fault tolerant systems. We describe here an experience in the application of formal methods to the design of fault-tolerant systems. The experience was done inside an applied research project. The input for our validation is an interaction policy between communicating objects, the Multiple Levels of Integrity policy, which has been defined within the project to enhance systems dependability.

The original definition of the policy simply consists of a set of declarative rules: it can be operationally realized defining a communication protocol. The protocol which carries out the integrity policy is formally specified as a collection of interacting processes in a process algebra. This formal specification is used as the basis for a formal validation of the protocol, exploiting model-checking. Specific interaction patterns, which subsume the most complex interaction schemata, are considered and temporal logic formulae expressing the non-violation of integrity rules are checked on them.

Keywords Integrity policies, fault tolerance, process algebras, model checking.

1 Introduction

The object–oriented paradigm has been established as the most pertinent basis to support the development of large computing and telecommunication systems, and several important international organizations are defining distributed object–oriented frameworks.

The software technology of fault tolerant systems is also moving towards this paradigm. Fault tolerant systems are a class of concurrent systems that are able to provide a service in spite of possible faults that may occur during the computation. The design of fault tolerant systems usually includes the modelling of faults and failures or the definition of fault tolerant schemata. Within the object–oriented paradigm, a fault tolerant schema usually describe a set of objects and their interactions.

An object that is part of a fault tolerant system is said to be *critical* if its failure can seriously affect the ability of the overall system to fulfil its safety requirements. Then, to enhance the dependability of fault tolerant systems, an important issue is to limit failure propagation among communicating objects. In particular, it is important to guarantee that a critical object (which should never fail - or should only fail with an extremely low probability) is not influenced by the failure of non critical ones.

One solution is to isolate critical components, dedicating to them, for instance, completely separated hardware resources with respect to those dedicated to the non critical ones, and to rigorously validate the critical components. However, complete isolation is not always possible because of the inevitable cooperation among the various parts of the system. In fact, the current trend is towards a greater integration of different functions on the same computer or network of computers. Another solution is to treat all the components as critical. The advantage is that communications do not have to be limited, the drawback is that all components have to be rigorously validated making this approach not always feasible due to the large dimensions of systems and to the use of "COTS" (Commercial Off The Shelf) components. Moreover, usually only a few components are really critical.

The issue of defining a fault tolerant schema to limit failure propagation can be solved with a compromise between the non–effectiveness of the first solution, and the high cost of the second one, thanks to the definition of particular interaction policies among objects, called *integrity policies* [1, 6, 18, 44, 45]. An integrity policy assigns a *level of integrity* to each component of an application. It is natural to assign a high integrity level to critical objects. Moreover, the policy states the communication patterns among pairs of components, depending on the respective integrity levels. Once a valuable policy has been adopted, not all the components need to be rigorously validated with the same effort, but only those which accomplish a critical task and those which provide data for these critical components.

Obviously, the policy itself should be seen as a critical component of the fault tolerant system that adopts it, and therefore it needs a rigorous validation too.

In the European GUARDS project, the *Multiple Levels of Integrity* policy has been defined within an object–oriented framework [44, 45]. This policy, thanks to OO features, is very flexible: objects are allowed to decrease their integrity level and are thus able to receive low level data. To prevent a downgrade of the whole system, it is sufficient to instantiate a new object capable (by means of proper data filtering algorithms) to restore a higher level of data integrity. The Multiple Levels of Integrity policy can be seen as a good representative of the application of the object–oriented paradigm to the definition of fault tolerant schemata.

This paper presents the rigorous validation of the Multiple Levels of Integrity policy by means of the application of formal methods and related support tools for its specification and verification.

Formal methods have already proved successful in specifying commercial and safety-critical software and in verifying protocol standards and hardware design [11, 21]. It is increasingly accepted that the adoption of formal methods in the life cycle development of systems guarantees higher levels of dependability even though formal methods cannot in general guarantee correctness. They can however greatly increase the understanding of a system by revealing, right from the earliest phases of the software development, inconsistencies, ambiguities and incompletenesses, which could cause subsequent faults. Moreover, international standards such as CENELEC EN50128 [16] recommend the use of "Formal Methods, including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B" in the software requirements specification and software design and implementation.

We show how the Multiple Levels of Integrity policy can be specified using a process algebra, and we show how the Multiple Levels of Integrity policy can be verified using model checking techniques, all in a context that is quite familiar to software engineers (OO systems, interaction policies). We also establish some guidelines for the application of the model–checking approach to the early verification of protocol scenarios other than the one considered.

Process algebras [31, 35] are formalisms that can describe a system consisting of communicating objects at a high level of abstraction. They are well suited to describing an interaction policy, since policies definition abstracts from the functionalities of the objects, and the relevant events to be described are the object invocations (the actions) which may change the object integrity level (the state).

Model checking algorithms [19, 20] have emerged as successful formal verification techniques. They have been defined to automatically check the truth of system properties, expressed as temporal logic formulae, on the finite state model representing the behavior of a system. Model checkers can easily be used by non–expert users as well. For this reason model checking has often been preferred in industries to other verification tools, and many efficient verification environments are currently available, based on model checking algorithms [14, 24, 32, 33, 17]. Model checking techniques have been already used to check the correctness of fault tolerant applications [2, 15, 13, 34, 43]. The main differences between the result presented here with respect to the previous ones, is that we deal with fault tolerance in an OO framework.

This work extends [28]. The added value of this paper is that we have defined, on the basis of the experience made in the GUARDS project and already presented in the former work, a methodology that can be applied more in general. Though, we have maintained the presentation of the approach strictly related to what was done inside the project, in order to substantiate the position that the approach is, and has been, concretely applicable. Moreover, in this revised version, we give a more general view of the verification process itself, and enhance the discussion on the generalization of the approach. We also add details on the validation experience which were not included in the conference paper. Finally, a large space is dedicated to discussion and related work.

This paper is organized as follows: in Sect. 2 we describe the *Multiple Levels of Integrity* policy, and present the JACK verification environment. In Sect. 3 we formally specify the integrity policy using the CCS/MEIJE process algebra. In Sect. 4 we present the model checking results and discuss the key points of the verification process. Finally, in Section 5 we deal with related works.

2 Background

The European project GUARDS (Generic Upgradable Architecture for Real–time Dependable Systems) has addressed the development of architectures, methods, techniques, and tools to support the design, implementation and validation of critical systems [39, 44]. The architecture produced in GUARDS has been designed to be instantiated to support different critical applications. It consists of COTS components as well as ad hoc defined mechanisms. The formal validation policy followed in GUARDS is based on the validation of selected critical components and mechanisms of the architecture, while COTS components are taken as already validated [38].

In particular, within the project, the Multiple Levels of Integrity policy has been defined [44, 45], and the verification environment JACK [4, 9] has been adopted to formally validate such a policy.

In section 2.1 we recall the definition of the GUARDS integrity policy, and in section 2.2 we present the formal validation framework. We focus on the CCS/MEIJE process algebra and the ACTL logic, so that the reader can understand the formal specification of the policy, and the formulae expressing the policy properties we then verify.

2.1 The Multiple Levels of Integrity policy

Integrity policies are defined to prevent failure propagation from non critical to critical components. They are based on the notion of integrity levels: a level of integrity, ranging over a finite set of natural values, is assigned to each component of an application. The use of integrity levels to select those critical components which need a rigorous validation, is also advocated by international standards. Indeed, international standards such as CENELEC EN50128 and RTCA–DO178B (EUROCAE ED–12B) [41] define integrity levels and discuss how these levels can be assigned to software components.

Failure propagation is avoided by integrity policies, which define some communication patterns to prevent corrupted data from reaching “high level” components. For instance, Biba’s policy [6], which is based on [1], forbids any flow of data from a low to a high integrity level. In [18] data can flow to a low level and go back, if it is possible to prove that they did not lose their integrity.

The *Multiple Levels of Integrity* policy, defined within an object–oriented framework, enhances the flexibility of the previous ones permitting some components to receive low level data [44, 45]. The integrity level of the components is decreased in these cases. To prevent a downgrade of integrity of the whole system, it is needed to instantiate a new object capable to restore a higher level of data integrity, by means of proper data filtering algorithms that validate the data (e.g. this can be performed using a majority vote on redundant data sources). The policy is based on the following concepts[1]:

- Each object O has an integrity level $il(O)$, ranging from 0, the lowest, to 3, the highest. Data are assigned the integrity level of the object which produced them. The number of levels has been chosen in GUARDS following the international standards mentioned above.
- *Single Level Objects* (SLO) are objects whose integrity level is fixed.
- *Multiple Level Objects* (MLO) are objects whose integrity level can be dynamically modified. An MLO of level 3 is allowed to receive data from an object of level 3, but also from an object of level 2 (or 1, or 0), in which case its level is decreased to 2 (1,0, resp.).
- *Validation Objects* (VO) are used to extract reliable data from low level objects.
- A set of *rules* is given, describing all the possible communication patterns among pairs of components, depending on the respective integrity levels.
- The communication model is based on the notion of method invocation. Method invocations are assigned an integrity level too. In particular, *read*, *write* and *read–write requests* are considered as abstractions of any method, with respect to the effect on the objects' states. The level of a write request corresponds to the level of the data which are written, the level of a read request corresponds to the minimum acceptable level of the data to be read. Read–write requests are assigned two integrity levels, one for read and one for write.

2.1.1 Single Level Objects

A Single Level Object is assigned a unique integrity level (**il**) which does not change during computations. Because of this, SLOs show a very restricted behaviour with respect to the integrity levels: an SLO of level n is only allowed to receive data from objects of level $\geq n$ and to send data to objects of level $\leq n$. In Fig. 1, an SLO with il=1 can receive data of a greater or equal level and produces data of level 1, which can thus be accepted as input by an SLO with

[1]Notice that the definition of the Multiple Levels of Integrity policy deals with objects, regardless of the fact that they may be instances of classes. The object model assumed by the policy is close to the CORBA one: indeed a trial implementation on CORBA was immediately possible [38].

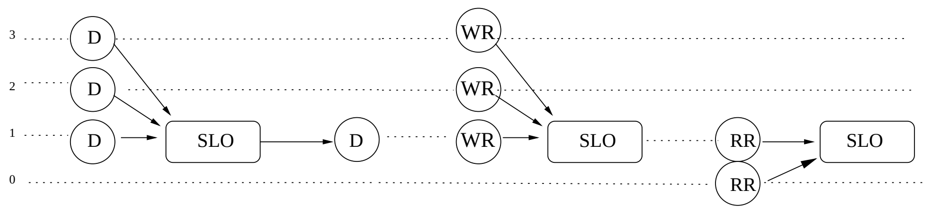

Figure 1: SLOs receive data (D) of a greater or equal level and produce data of their own level. Therefore, they accept write requests (WR) with a greater or equal integrity level and they accept read requests (RR) with a smaller or equal integrity level.

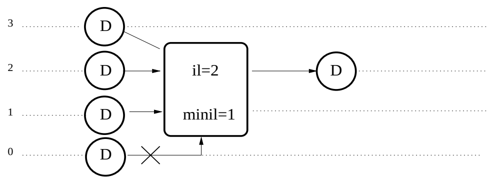

Figure 2: MLOs receive data (D) of a level greater or equal to *minil* and produces data of level *il*.

il=1 and an SLO with *il*=0. Consequently, an SLO with il=1 can accept write requests with greater or equal integrity level, and read requests with a lower or equal integrity level.

2.1.2 Multiple Level Objects

Multiple Level Objects are the core of the Multiple Levels of Integrity policy, since they are allowed to receive low level data. An MLO is assigned three integrity levels:

maxil which represents the maximum integrity level that the MLO can have. It is also called the *intrinsic level* of the MLO, since it is assigned during the design of the application. It is a statically defined value which does not change at execution time.

minil which represents the minimum value the integrity level of the MLO can reach while interacting with other objects. Unlike *maxil*, this is not a statically defined value: every time the MLO is invoked, a value to *minil*

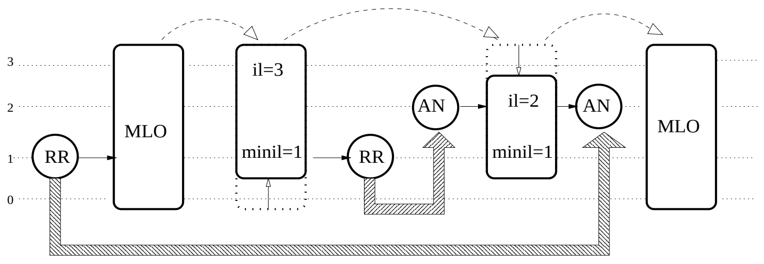

Figure 3: Behaviour of an MLO: dotted arrows follow the MLO's evolution, thick arrows bind requests to the corresponding answers.

is assigned. This value depends on the level of the invocation, and it can change (only to increase) during the computation to serve the request. No memory of it is kept after the answer to the invocation is returned: *minil* is local to an invocation.

il which is the current integrity level. It is set at invocation time to a value among *maxil* and *minil* and decreases if lower level data are received during the computation to serve the invocation. As for *minil*, *il* is local to each invocation and no memory of its value is kept between distinguished invocations.

The policy requires a new MLO instance to be created every time the MLO is invoked, in order not to rapidly downgrade the integrity level of the whole system. The values for *il* and *minil* taken by the new instance only depend on the level of the invocation. As a consequence, an MLO cannot be used to implement a component which has to store some data. This means that an MLO, from a functional point of view, is a stateless object: only SLOs can store data.

In Fig. 2, we show that an MLO can receive data with an integrity level greater or equal to its "minil", and produce data with an integrity level equal to its "il". In Fig. 3, we provide an example of the evolution of an MLO in response to an invocation: when an MLO with *maxil* $= 3$ receives a read request of level 1, it sets its *minil* to 1 to "remember" that no answer with integrity level smaller than 1 can be returned. The value of *il* still equals *maxil*: in fact, a read request does not corrupt the integrity level of the MLO. Suppose the MLO needs to delegate part of the answer construction, sending another read request to a third object. The level assigned to the request equals *minil*: an answer to this request is accepted if greater or equal to *minil*, as in this case. Since the integrity level of the answer is 2, the MLO can accept it but *il* is decreased to level 2. Finally, an answer to the first request is provided, whose level equals

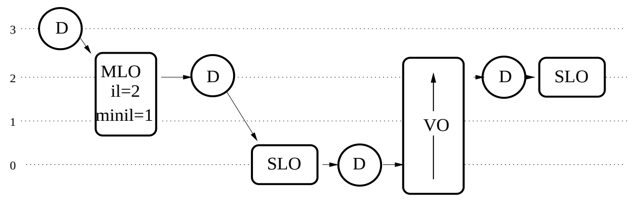

Figure 4: A VO is able to (partly) restore the integrity of data.

the current *il*, and the MLO restores its initial state, i.e. its maximum integrity level is again 3.

2.1.3 Validation Objects

In real systems, it is sometimes necessary to get data from unreliable sources, such as sensors, and use them in critical tasks. However, this use could either lower the level of the entire system or violate the integrity policy. We thus need a safe way to upgrade the integrity level of these data.

Validation Objects (VO) are special kinds of objects used to extract reliable data from low level objects. An example of a Validation Object is the one that uses a redundant number of objects as a source for the data, and performs a majority voting on them.

From the point of view of the integrity policy, Validation Objects are SLOs, since they provide information at a fixed level of integrity. An example is provided in Fig. 4: some data are corrupted since they are manipulated by objects (an SLO and an MLO) with a low integrity level. The VO manages to (partially) restore the integrity of the data. In this example the VO is able to return data at level 2.

2.1.4 Rules for communications between objects

The policy defines a set of rules to constrain communication among pairs of components, depending on their integrity level. We list them in Table 1 considering all the possible combinations of invocations. In the table and in the following, we call A and B the invoking and the invoked objects, respectively.

The first part of the table considers invocation conditions. The invocation is refused if the specified condition is not satisfied. If it is accepted, the invoked object might have to change its integrity level, as shown in the second part of the table, where invocation effects are considered. We only treat the cases in which the invoked object is an MLO, since only MLOs can update their level.

In the case of read or read–write invocation, an answer is returned at the end of the method execution. To model the effect of the answer on the invoking object, there is a last rule, which is not in the table, since it only applies in one case and does not deserve a tabular presentation. If the invoking object

Conditions	$A\&B$ SLOs	A SLO, B MLO	A MLO, B SLO	$A\&B$ MLOs
A reads B	$il(A) \leq il(B)$	$il(A) \leq maxil(B)$	$minil(A) \leq il(B)$	$minil(A) \leq maxil(B)$
A writes B	$il(B) \leq il(A)$	$always$	$il(B) \leq il(A)$	$always$
A r-w B	$il(A) = il(B)$	$il(A) \leq maxil(B)$	$minil(A) \leq il(B) \leq il(A)$	$minil(A) \leq maxil(B)$

Effect	A SLO, B MLO	$A\&B$ MLOs
A reads B	$minil(B) := il(A)$; $il(B) := maxil(B)$	$minil(B) := minil(A)$; $il(B) := maxil(B)$
A writes B	$il(B) := min(il(A),\ maxil(B))$	$il(B) := min(il(A),\ maxil(B))$
A r-w B	$minil(B), il(B) := il(A)$	$minil(B) := minil(A)$; $il(B) := min(il(A),\ maxil(B))$

Table 1: Conditions to be satisfied for a method invocation to be accepted, and the effect on the level of objects after acceptance.

was an MLO, then the returned data may decrease its integrity level as follows: $il(A) := min(il(A), il(B))$.

2.1.5 An Example

As an example of application of the policy, we consider a safety-critical system controlling a chemical plant: the data on which the control law operates are high integrity data. The control algorithm itself has a high integrity level. Part of the data is from time to time stored to enable off-line analysis of the production of the plant. The storing and analysis functions, though important for the productivity of the plant, are not safety critical, and hence have a lower integrity level. In principle, they could be isolated as much as possible, confining them for example on a separate computer. However, suppose that the control and the analysis functions use internally the same integration algorithm, the first one on *vital* data, the second on data which is also entered from the operator, which therefore cannot be trusted. All the identified functions can be hosted on the same computer, provided that they do not violate the integrity policy that forbids the lower integrity functions to write on the vital control data. Note that the integration function could now be used from both the analysis and the control algorithms. The object structure of this example system could be:

– A high integrity SLO implementing the control algorithm;

- A low integrity SLO implementing the storing function;
- A low integrity SLO implementing the analysis function;
- A high integrity MLO implementing the integration algorithm.

The integration algorithm (note that this is a stateless object) would lower its integrity level when invoked by the analysis function to access non-critical data.

Only the high integrity SLO and MLO above need to undergo a thorough validation.

2.2 Formal Validation Methodology in GUARDS

All the critical mechanisms (i.e. Inter–Consistency mechanism, Fault–Treatment mechanism, and Multiple Levels of Integrity policy) of the GUARDS architecture have been validated according to the following steps [3]:

- Formal specification of the mechanism using the CCS/MEIJE process algebra [10].

  Process algebras are based on a simple syntax and are provided with a rigorous semantics defined in terms of Labelled Transition Systems (LTSs). LTSs describe the behaviours of a system as sequences of elementary actions, where each action is seen as a state transformer.

- Use of the ACTL temporal logic [23] to describe the properties that express the desired behaviour of the mechanism.

  Temporal logics have been proposed [23, 27] to provide a means to express properties of concurrent systems at a high level of abstraction. The logic ACTL is a branching-time temporal logic whose interpretation domains are LTSs.

- Generation of the (finite state) model of the mechanism.

  To this end, we use the tools of the JACK (*Just Another Concurrency Kit*) verification environment [4, 9]. JACK is a formal specification and verification environment based on the use of process algebras, LTSs and temporal logic formalisms, supporting many phases of the systems development process.

- Model checking of the ACTL formulae against the model of the mechanism, using the efficient model checker for ACTL available in JACK, AMC [22].

  Model checking is an automated verification method for checking finite state systems against properties specified in temporal logic [19, 20]. The proof of the properties is carried out by means of an exhaustive search on the complete behaviour (model) of the system.

The choice of CCS and JACK was made inside the GUARDS project also to have a testbed on which to test and refine the verification tools themselves. Indeed, CCS is one of the formal methods listed in the EN50128 standard, so its use

$a : P$	Action prefix	Action a is performed, and then process P is executed. Action a is in Act_τ
$P + Q$	Nondeterministic choice	Alternative choice between the behavior of process P and that of process Q
$P \parallel Q$	Parallel composition	Interleaved executions of processes P and Q. The two processes synchronize on complementary input and output actions (i.e. actions with the same name but a different suffix)
$P \setminus a$	Action restriction	The action a can only be performed within a synchronization

Table 2: A fragment of CCS/MEIJE syntax

was positively considered in an industrial context which looks compliance to that standard (one of the partners of the GUARDS project was indeed a railway signalling company): the use of a basic formalism such as CCS, directly related to a finite state representation, is in line with the conservativeness of safety critical software industries. Obviously, other formalisms equipped with powerful verification tools could have been used as well following the same validation process.

2.2.1 The CCS/MEIJE process algebra

Process algebras [31, 35] rely on a small set of basic operators, which correspond to primitive notions of concurrent systems, and on one or more notions of behavioral equivalence or preorder. Behavioral equivalences are used to study the relationships between descriptions of the same system at different levels of abstraction (e.g., specification and implementation).

The process algebra we have used is CCS/MEIJE [10]. In CCS/MEIJE a system consists of a set of communicating processes. Each process executes input and output actions, and synchronizes with other processes to carry out its activities. The CCS/MEIJE syntax is based on a finite set *Act* of atomic action names. Such names represent output actions if they are terminated by "!", or input ones if they are terminated by "?". Moreover, τ denotes the special action not belonging to *Act*, representing the unobservable action (to model internal process communications). We assume $Act_\tau = Act \cup \{\tau\}$.

The syntax of CCS/MEIJE processes is based on a set of operators that allow complex processes to be built from simpler ones. The syntax permits a two-layered design of *process terms*. The first level is related to *sequential regular terms*, the second one to *networks* of parallel sub-processes supporting communication and action renaming or restriction. In Table 2 we present the subset of the CCS/MEIJE operators we will use in the following.

The semantic models of CCS/MEIJE terms are Labelled Transition Systems which describe the behavior of a process in terms of states, and labelled transitions, which relate states. An LTS is a 4–tuple $\mathcal{A} = (Q, q_0, Act_\tau, \rightarrow)$, where: Q is a finite set of states; q_0 is the initial state; $\rightarrow \subseteq Q \times Act_\tau \times Q$ is the transition relation. In Table 3, we provide the structural operational semantics of the

$a : P$	$a : P \xrightarrow{a} P$		
$P + Q$	$\dfrac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$	$\dfrac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$	
$P \parallel Q$	$\dfrac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}$	$\dfrac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$	$\dfrac{P \xrightarrow{a?} P', Q \xrightarrow{a!} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$
$P \setminus a$	$\dfrac{P \xrightarrow{b} P'}{P \setminus a \xrightarrow{b} P' \setminus a}\ b \neq a$		

Table 3: Structural operational semantics of the considered CCS/MEIJE operators, in terms of LTSs

considered CCS/MEIJE operators, in terms of LTSs [35].

2.2.2 The ACTL temporal logic

We introduce here the branching time temporal logic ACTL [23], which is the action based version of CTL [27]. ACTL is well suited to expressing the properties of a system in terms of the actions it performs at its working time. In fact, ACTL embeds the idea of "evolution in time by actions" and is suitable for describing the various possible temporal sequences of actions that characterize a system's behavior.

The syntax of ACTL is given by the following grammar, where ϕ denotes a state property:

$$\phi ::= \mathit{true} \mid {\sim}\phi \mid \phi \ \&\ \phi' \mid [\mu]\phi \mid AG\ \phi \mid A[\phi\{\mu\}U\{\mu'\}\phi' \mid E[\phi\{\mu\}U\{\mu'\}\phi']$$

In the above rules μ is an action formula defined by:

$$\mu ::= \mathit{true} \mid a \mid \mu \vee \mu \mid {\sim}\mu \qquad \text{for } a \in \mathit{Act}$$

Labelled transition systems are the interpretation domains of ACTL formulae. We provide here an informal description of the semantics of ACTL operators. The formal semantics is given in [23].

Any state satisfies *true*. A state satisfies ${\sim}\phi$ if and only if it does not satisfy ϕ; it satisfies $\phi\ \&\ \phi'$ if and only if it satisfies both ϕ and ϕ'. A state satisfies $[a]\phi$ if for all next states reachable with a, ϕ is true. The meaning of $AG\ \phi$ is that ϕ is true now and *always* in the future.

A state P satisfies $A[\phi\{\mu\}U\{\mu'\}\phi']$ if and only if in each path exiting from P, μ' will eventually be executed. It is also required that ϕ' holds after μ', and all the intermediate states satisfy ϕ; finally, before μ' only μ or τ actions can be executed. The formula $E[\phi\{\mu\}U\{\mu'\}\phi']$ has the same meaning, except that it requires one path exiting from P, and not all of them, to satisfy the

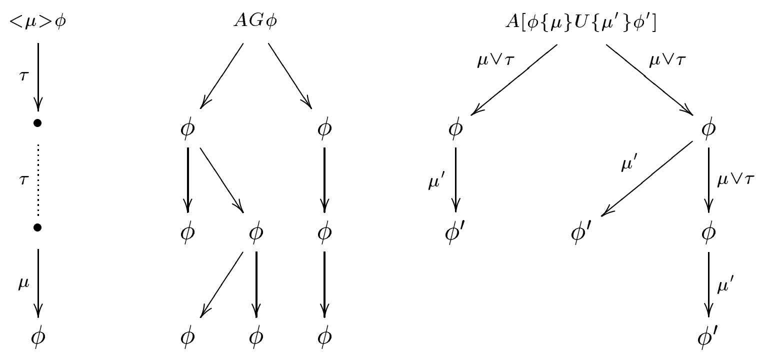

Figure 5: Models and ACTL formulae.

given constraint. A useful formula is $A[\phi\{true\}U\{\mu'\}\phi']$ where the first action formula is *true* this means that any action can be executed before μ'.

Some derived operators can be defined: $\phi \mid \phi'$ stands for $\sim(\sim\phi \ \& \ \sim\phi')$; $<\mu>\phi$ stands for $\sim[\mu]\sim\phi$; finally, $EF\phi$ stands for $\sim AG\sim\phi$ (this is the *eventually* operator, whose meaning is that ϕ will be true sometime in the future).

In Figure 5 we exemplify the truth of some formulae on some models.

Example. The ACTL logic may express *safety* and *liveness* properties in terms of the actions a system can perform. Safety properties claim that nothing bad happens; i.e., that there is no path in the LTS in which a given action sequence occurs. Liveness properties claim that something good eventually happens; i.e., that there exists a path in the LTS in which a given action sequence occurs. In this setting, for example, the formula:

$$AG[a]A[true\{false\}U\{b\}true]$$

means that, in all paths of the LTS, any action a has to be immediately followed by an action b. The formula well characterizes safety requirements. On the other hand, a typical ACTL formula stating a liveness property is, for example:

$$EF\ <b>\ true$$

which means that on some path of the LTS, action b will eventually be executed.

3 Formal specification of the Integrity policy

We present in this section the formal description of the GUARDS integrity policy using the CCS/MEIJE process algebra.

Our claim is that process algebras are particularly suited to describing the Multiple Levels of Integrity policy. In fact, the policy definition abstracts from the functionality of the objects, and the actions are the relevant events we need to describe, i.e. method invocations and method returns (i.e. answers). The actions may change the object integrity level, which is our abstraction of the state. We model method invocation through a remote procedure call, in which the invoking object waits for the method return event.

The formal description of the policy is given on the basis of the rules in Section 2.1.4. Note that these rules cannot be formalized within a static framework, such as traditional type theory, due to the presence of dynamically changing integrity levels for MLOs. We rather need to associate a process to each object, to model its dynamic behaviour with respect to the integrity level.

We first need to define the process variables and the set of actions performed by SLOs, MLOs, and VOs. We consider here only the action labels, abstracting from their possible role as input or output actions.

$\mathtt{SLO}_{\mathtt{x}}$, $\mathtt{MLO}_{\mathtt{x}}$, and $\mathtt{VO}_{\mathtt{x}}$ are process variables denoting the processes defining a Single Level Object with integrity level `x`, a Multiple Level Object with the statically defined *maxil* value `x`, a Validation Object providing data of level `x`, respectively.

$\mathtt{read\_request}_{\mathtt{x}}$ is a read request action of level `x`. This means that the invocation was issued either by an SLO with `x` as *il* or by an MLO with `x` as *minil*.

$\mathtt{answer}_{\mathtt{x}}$ is an answer action. Value `x` can be the current *il* of the object which is answering, or -1: $\mathtt{answer}_{\mathtt{-1}}$ means something like: "I cannot answer" (we use this notation for the sake of uniformity).

$\mathtt{write\_request}_{\mathtt{x}}$ denotes a write request issued by an object with `x` as *il*. We call `x` the level of the write request. Write requests are not answered.

$\mathtt{read\_write\_request}_{\mathtt{x},\mathtt{y}}$ denotes a read–write request issued either by an MLO with `x` as *minil* and `y` as *il* or by an SLO with $il = \mathtt{x} = \mathtt{y}$. Variable `x` denotes the read level of the request, variable `y` denotes the write level.

We first describe the specification of an SLO. In Sects. 3.2 and 3.3 we will describe MLOs and VOs.

3.1 Formal description of the behaviour of SLO$_x$

We give in the following the specification of an SLO with integrity level equal to `x` as a CCS/MEIJE mutually recursive process.

```
SLOx =  read_request_y? (case y≤x) : SAT_R_REQ_Sx                    +
        read_request_y? (case y > x) : answer_-1! : SLOx               +
        write_request_y? (case y < x) : SLOx                           +
        write_request_y? (case y ≥ x) SAT_W_REQ_Sx                     +
        read_write_request_y,z? (case y ≤ z < x) : answer_-1!: SLOx    +
        read_write_request_y,z? (case y ≤ x ≤ z) : SAT_R_REQ_Sx        +
        read_write_request_y,z? (case y > x) answer_-1! : SLOx
```

where `read_request_y? (case y≤x) : P` is a shorthand for the process:
`read_request_0?:P + read_request_1?:P +...+ read_request_x?:P`.

We will use this kind of shorthand throughout. Indeed, since the relevant decisions only depend on the relative value of the level of the interacting parties, we can describe these processes using the inequalities $y \leq x$, ... to collapse sets of equivalent behaviours into one line of description.

When a read request of level y is received and $y \leq x$, then the SLO makes the needed computation to serve the request. On the contrary, a read request of level greater than x cannot be considered, since the SLO has not the needed integrity level to supply an answer.

When a write request is received, a computation can be performed to satisfy the requests, but no answer is due. If the level of the request is smaller than x, then it is ignored, and no computation is performed.

A read–write request is dealt with in the same way as the composition of read and write ones: it is accepted only if the read level y is smaller or equal than x and the write level z is greater or equal than x.

With SAT_R_REQ_Sx we represent an SLO$_x$ which satisfies a read or a read–write invocation. Its specification is:

```
SAT_R_REQ_Sx =
    answerx! : SLOx                                       +     (1)
    write_request_x! : SAT_R_REQ_Sx                       +     (2)
    read_request_x! : (
       answery? (case y < x) : answer_-1! : SLOx +              (3)
       answery? (case y ≥ x) : SAT_R_REQ_Sx )             +     (4)
    read_write_request_x,x! : (
       answery? (case y < x) : answer_-1! : SLOx +              (5)
       answery? (case y ≥ x) : : SAT_R_REQ_Sx )                 (6)
```

Indeed, both to satisfy a read and a read–write request, the SLO can:

– provide the answer to the caller. In this case the SLO has ended its duty. The answer carries the integrity level of the SLO: this is necessary if the invoking object is another MLO that might need to update its *il*; (1)

– send a write request to another object and continue; (2)

– send a read or a read–write request, wait for the answer. Continuation depends on the level of the answer received, if this is too low, then the computation is stopped, and a "I cannot answer your request" message is sent; (3)(4)(5)(6)

The specification is non–deterministic, and the SLO might get into a loop and never send an answer back. This happens if steps (1), (3), or (6) are never taken. Non–determinism is a consequence of the abstraction from the object functionalities. In particular, step number (1), which is the normal loop exit, is taken or not depending on the functional behaviour of the object , and it is correct that, if no integrity level violation occurs (in this case step (3) or (6) are taken), the overall behaviour of the object depends only on its functional description.

With $\texttt{SAT\_W\_REQ\_S}_{\texttt{x}}$ we represent an SLO_x which satisfies a write invocation. The description of $\texttt{SAT\_W\_REQ\_S}_{\texttt{x}}$ can be immediately derived from $\texttt{SAT\_R\_REQ\_S}_{\texttt{x}}$. Indeed the behaviour of $\texttt{SAT\_W\_REQ\_S}_{\texttt{x}}$ corresponds to that of $\texttt{SAT\_R\_REQ\_S}_{\texttt{x}}$, where all actions of the type $\texttt{answer}_{\texttt{il}}\texttt{!}$ are removed, since no answer is due to write requests.

```
SAT_W_REQ_S_x =
      SLO_x                                        +
      write_request_x! : SAT_W_REQ_S_x             +
      read_request_x! : (
         answer_y? (case y< x) : SLO_x +
         answer_y? (case x≤y) : SAT_W_REQ_S_x )    +
      read_write_request_x,x! : (
         answer_y? (case y<x) : SLO_x +
         answer_y? (case x≤y) : SAT_W_REQ_S_x )
```

3.2 Formal description of the behaviour of MLO_x

The description of MLOs is more complex than that of SLOs. One particular difference arises especially when dealing with write requests: while an SLO ignores a write request with a low integrity level, an MLO always accepts a method invocation corresponding to a write request (decreasing *il* to the level of the request).

```
MLO_x =  read_request_y? (case y≤x) : SAT_R_REQ_y,x,x              +
         read_request_y? (case y>x) : answer_-1! : MLO_x           +
         write_request_y? (case y≤x) : SAT_W_REQ_0,y,x             +
         write_request_y? (case y>x) : SAT_W_REQ_0,x,x             +
         read_write_request_y,z? (case y≤z≤x) : SAT_R_REQ_y,z,x    +
         read_write_request_y,z? (case y≤x≤z) : SAT_R_REQ_y,x,x    +
         read_write_request_y,z? (case y>x) : answer_-1! : MLO_x
```

When a read request of level `y` is received and $\mathtt{y}\leq\mathtt{x}$, then the MLO makes the necessary computation to serve the request taking `y` as its *minil* value, and `x`, its *maxil* value, as its value for *il*, and behaves like $\mathtt{SAT\_R\_REQ}_{\mathtt{y,x,x}}$. On the contrary, a read request of level greater than x cannot be considered, since the MLO does not have the integrity level needed to supply an answer.

When a write request is received, the MLO takes 0 as *minil*, and the minimum among `x` (its *maxil*) and `y` (the level of the request) as *il*. A computation can then be performed to satisfy the requests, but no answer is due.

A read–write request is dealt with as the composition of a read and a write one. Depending on its read and write values, the object can refuse the invocation, and behave as $\mathtt{SAT\_R\_REQ}_{\mathtt{y,z,x}}$ or as $\mathtt{SAT\_R\_REQ}_{\mathtt{y,x,x}}$.

Processes $\mathtt{SAT\_R\_REQ}_{\mathtt{y,z,x}}$ and $\mathtt{SAT\_R\_REQ}_{\mathtt{y,x,x}}$ are obtained by instantiating the following parametric definition of $\mathtt{SAT\_R\_REQ}_{\mathtt{min,il,max}}$. The CCS/MEIJE description of process $\mathtt{SAT\_R\_REQ}_{\mathtt{y,x,x}}$ is obtained, for instance, by substituting `y` for `min`, and `x` for both `il` and `max` in $\mathtt{SAT\_R\_REQ}_{\mathtt{min,il,max}}$.

$\mathtt{SAT\_R\_REQ}_{\mathtt{min,il,max}}$ represents an MLO which satisfies a read invocation: the MLO is characterized by the three values `min, il`, and `max`. We recall that, when satisfying an invocation, the MLO can change these values, depending on the integrity level of other objects it communicates with during the computation. When the request has been satisfied, the MLO forgets the `min` and `il` values and keeps only its intrinsic integrity level `max` which has never changed during the computation. This is possible since we made the assumption that a new object instance is created every time the object is invoked, and thus the object keeps no memory of previous invocations.

```
SAT_R_REQ$_{\mathtt{min,il,\ max}}$ =
    answer$_{\mathtt{il}}$! : MLO$_{\mathtt{max}}$                                     +
    write_request$_{\mathtt{il}}$! : SAT_R_REQ$_{\mathtt{min,il,max}}$                 +
    read_request$_{\mathtt{min}}$! : (
        answer$_{\mathtt{x}}$? (case x<min) : answer$_{\mathtt{-1}}$! : MLO$_{\mathtt{max}}$ +
        answer$_{\mathtt{x}}$? (case min$\leq$x$\leq$il) : SAT_R_REQ$_{\mathtt{min,x,max}}$ +
        answer$_{x}$? (case x$\geq$il) : SAT_R_REQ$_{\mathtt{min,il,\ max}}$)         +
    read_write_request$_{\mathtt{min,il}}$! : (
        answer$_{\mathtt{x}}$? (case x<min) : answer$_{\mathtt{-1}}$! : MLO$_{\mathtt{max}}$ +
        answer$_{\mathtt{x}}$? (case min$\leq$x$\leq$il) : SAT_R_REQ$_{\mathtt{min,x,max}}$ +
        answer$_{\mathtt{x}}$? (case x$\geq$il) : SAT_R_REQ$_{\mathtt{min,il,\ max}}$)
```

The behaviour of $\mathtt{SAT\_W\_REQ}_{\mathtt{0,y,x}}$ and $\mathtt{SAT\_W\_REQ}_{\mathtt{0,x,x}}$ corresponds to that of $\mathtt{SAT\_R\_REQ}_{\mathtt{0,y,x}}$ and $\mathtt{SAT\_R\_REQ}_{\mathtt{0,x,x}}$ where all actions of the type $\mathtt{answer}_{\mathtt{il}}\mathtt{!}$ are removed, as we have done in the case of SLOs. We omit their definition.

3.3 Formal description of the behaviour of VO$_x$

A validation object provides data at a fixed integrity level, that is the level of integrity to which it is able to raise data. It only accepts read requests, and behaves as follows:

$$\begin{aligned}\mathtt{VO_x} = \;& \mathtt{read\_request_y?\ (case\ y{\le}x)\ :\ SAT\_R\_REQ\_V_x} \quad + \\ & \mathtt{read\_request_y?\ (case\ y{>}x)\ :\ answer_{-1}!\ :\ VO_x}\end{aligned}$$

In the above definition the validation object can try to satisfy the request, or it answers immediately that this is not possible. With $\mathtt{SAT\_R\_REQ\_V_x}$ we represent a VO$_x$ which satisfies a read invocation, defined as in the following:

$$\mathtt{SAT\_R\_REQ\_V_x} = \mathtt{answer_x!\ :\ VO_x} + \mathtt{answer_{-1}!\ :\ VO_x}$$

i.e. a request satisfaction can fail as well. This is the case, for instance, if the VO cannot find all the (redundant) data it needs.

4 Validation of the Integrity policy

The Multiple Levels of Integrity policy has to guarantee that the interaction among different components does not affect the overall confidence of the application, i.e that a non–critical component does not corrupt a critical one. In particular, data of a low integrity level cannot flow to a higher integrity level (unless through a Validation Object, which is the only kind of object authorized to break this rule). This condition should hold for isolated objects and in any schema of interaction among objects.

In particular, we address nested and concurrent invocations as validation cases, since in an object–oriented framework most schemata can be reduced to combinations of these two interaction patterns. We show in the following some properties which are sufficient to ensure that any interaction schema is not erroneously violated when read requests are considered. The properties that deal with write and read–write requests can be similarly expressed following the rules of Section 2.1.4.

Prop 1 An object with intrinsic level i cannot provide answers of level $j > i$.

Prop 2 An object with intrinsic level i does not accept read requests of level $j > i$.

Prop 3 If an MLO with intrinsic level i receives a read request of level $j \leq i$, and, to serve the request, it invokes with a read request a third object of intrinsic level *maxil* smaller than j, then it cannot answer the initial request. Indeed, its level is decreased to the *maxil* value of the third object because of the new data received.

Prop 4 If an MLO with intrinsic level i receives a read request of level $j \leq i$, and then a write request of level $k < j$, then it can still answer the read request. In other words, its level is not decreased by the concurrent invocation.

We will check these properties against the model of an isolated object or the model of a combination of MLOs based on nested or concurrent invocation. Indeed, the most interesting cases are those involving MLOs, which can change their integrity levels during the computation.

The above properties are first formalized as ACTL formulae, then the AMC model checker is used to verify their satisfiability on the model of the selected subsystems.

Since actual validation by model–checking requires non–parametric models, we will define particular instances of the considered validation cases, which will be shown to be sufficiently representative to be generalized. Hence, we will use these instances to prove the set of temporal logic formulae expressing the integrity properties above, and then we will discuss how model checking results can be generalized. For example, the case of nested invocations is generalized in order to prove a more general variant of **Prop 3**:

Prop G–3 If an MLO with intrinsic level i receives a read request of level $j \leq i$, and, to serve the request, it starts a chain of nested invocations to other objects, one of which has intrinsic level $k < j$, then it cannot answer the initial request.

4.1 Isolated object

Multi Level Objects (MLOs) have already been identified as the most complex and interesting ones, and are the core of the innovative integrity policy. Hence, we concentrate on them to show the validity of Prop 1 and 2 above.

In particular, we consider a system consisting only of the object A_2, which is an MLO with *maxil* equal to 2. In this case, Prop 1 and 2 can be expressed by the following ACTL formulae[2]:

F1: $\sim EF\ \langle A_2\_answer_3! \rangle\ true$

F2: $AG\ [A_2\_read\_request_3?]\ A\ [true\ \{false\}\ U\ \{A_2\_answer_{-1}!\}\ true]$

i.e. A_2 cannot provide answers of level 3, nor serve read requests of level 3.

4.1.1 Model–checking results and generalization

Formulae F1 and F2 have been proven true on the model of A_2 using the model checker AMC for ACTL. The process algebraic description of A_2 has a low complexity, and the LTS has a really small number of transitions and states (see Table 4).

For the other values (i.e. 0, 1, 3) of *maxil*, we can repeat the verification using the corresponding formulae and processes. To provide an intuition of how the formalization of properties Prop 1 and 2 looks like for the other values of

[2]In the following we rename requests and answers: requests carry the name of the object which is invoked to serve the request, answers take the name of the answering object.

maxil, we provide the general formulae and instantiate them to the various cases. Here and in the rest of this section, we will use some shorthand to express the general formulae: for instance $\underset{j>1}{\&}\ \phi_j$ stays for ϕ_2 & ϕ_3.

Prop 1 says that any A_i satisfies:

$$\underset{j>i}{\&} \sim EF\langle A_i\_answer_j!\rangle true.$$

F1 is the instantiation of this formula for A_2, the other instances are:

A_0: $\sim EF\ \langle A_0\_answer_1!\rangle\ true\ \&\ \sim EF\ \langle A_0\_answer_2!\rangle\ true\ \&$
$\sim EF\ \langle A_0\_answer_3!\rangle\ true$
A_1: $\sim EF\ \langle A_1\_answer_2!\rangle\ true\ \&\ \sim EF\ \langle A_1\_answer_3!\rangle\ true$
A_3: $true$

Prop 2 says that any A_i satisfies:

$$\underset{j>i}{\&}\ AG[A_i\_read\_request_j?]A[true\{false\}U\{A_i\_answer_{-1}!\}true]$$

F2 is the instantiation of this formula for A_2, the other instances are:

A_0: $AG\ [A_0\_read\_request_1?]\ A\ [true\ \{false\}\ U\ \{A_0\_answer_{-1}!\}\ true]\ \ \&$
$AG\ [A_0\_read\_request_2?]\ A\ [true\ \{false\}\ U\ \{A_0\_answer_{-1}!\}\ true]\ \ \&$
$AG\ [A_0\_read\_request_3?]\ A\ [true\ \{false\}\ U\ \{A_0\_answer_{-1}!\}\ true]$
A_1: $AG\ [A_1\_read\_request_2?]\ A\ [true\ \{false\}\ U\ \{A_1\_answer_{-1}!\}\ true]\ \ \&$
$AG\ [A_1\_read\_request_3?]\ A\ [true\ \{false\}\ U\ \{A_1\_answer_{-1}!\}\ true]$
A_3: $true$

4.2 Nested invocations

We take into account here the case in which an MLO of a given level, in response to a read–request, invokes with a read–request another MLO of a lower integrity level. This is indeed the most complex case of nested invocations: all the other combinations (SLO vs. MLO, read–request vs. write–request or read/write–request, different levels of integrity) can be reduced to this one: in any case, a separate validation by model–checking of these other cases can be made following what is presented here.

We describe such a system with the parallel composition of the two objects:

- `A2` which is an MLO with *maxil* 2
- `B0` which is an MLO with *maxil* 0

To clarify the behaviour of the objects in case of nested invocations, we consider a possible scenario, depicted in Fig. 6. Let a read request of level 1 be received by A_2 and assume that the functional description of A_2 imposes that, to serve such a request, a further read request must be sent to B_0. We list the steps of such an execution of $\texttt{A}_2 \parallel \texttt{B}_0$:

1. at the beginning we have: $\texttt{A}_2 \parallel \texttt{B}_0$;
2. $\texttt{A}_2$ receives the read request performing action $\texttt{A}_2\texttt{\_read\_request}_1\texttt{?}$;
3. the request is accepted, since request_level $= 1 \leq$ maxil($\texttt{A}_2$) $= 2$;
4. we now have: $\texttt{SAT\_R\_REQ}_{1,2,2} \parallel \texttt{B}_0$;
5. $\texttt{B}_0\texttt{\_read\_request}_1\texttt{!}$ is output by $\texttt{A}_2$ (and received by $\texttt{B}_0$);
6. the request cannot be accepted by $\texttt{B}_0$ since:

   minil($\texttt{SAT\_R\_REQ}_{1,2,2}$) $=$ request_level $= 1 >$ maxil($\texttt{B}_0$) $=0$;
7. we thus now have: ($\texttt{A}_2$ waiting for an answer) $\parallel$ $\texttt{B}_0\texttt{\_answer}_{-1}\texttt{!} : \texttt{B}_0$ where " $\texttt{A}_2$ waiting for an answer" is equivalent to:

   $\texttt{B}_0\texttt{\_answer}_{-1}\texttt{?} : \texttt{A}_2\texttt{\_answer}_{-1}\texttt{!} : \texttt{A}_2 \;+\; \texttt{B}_0\texttt{\_answer}_0\texttt{?} : \texttt{A}_2\texttt{\_answer}_{-1}\texttt{!} : \texttt{A}_2 \;+$
   $\texttt{B}_0\texttt{\_answer}_1\texttt{?} : \texttt{SAT\_R\_REQ}_{1,1,2} \;+\; \texttt{B}_0\texttt{\_answer}_2\texttt{?} : \texttt{SAT\_R\_REQ}_{1,2,2} \;+$
   $\texttt{B}_0\texttt{\_answer}_3\texttt{?} : \texttt{SAT\_R\_REQ}_{1,2,2}$
8. $\texttt{B}_0\texttt{\_answer}_{-1}\texttt{!}$ is output by $\texttt{B}_0$ (and received by $\texttt{A}_2$);
9. we thus have: $\texttt{A}_2\texttt{\_answer}_{-1}\texttt{!} : \texttt{A}_2 \parallel \texttt{B}_0$;
10. $\texttt{A}_2\texttt{\_answer}_{-1}\texttt{!}$ is output by $\texttt{A}_2$ to acknowledge that it cannot satisfy the request it was trying to serve;
11. we end up back in the initial configuration: $\texttt{A}_2 \parallel \texttt{B}_0$

The behaviour corresponding to this scenario is:

$$\begin{aligned}
&\texttt{A}_2 \parallel \texttt{B}_0 \xrightarrow{\texttt{A}_2\texttt{\_read\_request}_1\texttt{?}} \texttt{SAT\_R\_REQ}_{1,2,2} \parallel \texttt{B}_0\\
&\xrightarrow{\tau} \underbrace{\ldots + \texttt{B}_0\texttt{\_answer}_{-1}\texttt{?:}\texttt{A}_2\texttt{\_answer}_{-1}\texttt{!:}\texttt{A}_2 + \ldots}_{\texttt{A}_2\ waiting\ for\ an\ answer} \parallel \texttt{B}_0\texttt{\_answer}_{-1}\texttt{!:}\texttt{B}_0\\
&\xrightarrow{\tau} \texttt{A}_2\texttt{\_answer}_{-1}\texttt{!:}\ \texttt{A}_2 \parallel \texttt{B}_0\\
&\xrightarrow{\texttt{A}_2\texttt{\_answer}_{-11}\texttt{!}} \texttt{A}_2 \parallel \texttt{B}_0
\end{aligned}$$

4.2.1 Properties in the case of nested invocations

Prop 3 in the case of $\texttt{A}_2 \parallel \texttt{B}_0$ can be expressed by the ACTL formula:

F3: $AG\ [A_2\_read\_request_1?]\ AG\ [\nu]\ A\ [true\ \{\mu\}\ U\ \{A_2\_answer_{-1}!\}\ true]$

with $\mu =\sim (A_2\_answer_0! \vee A_2\_answer_1! \vee A_2\_answer_2! \vee A_2\_answer_3!)$ and $\nu = B_0\_read\_request_0! \vee B_0\_read\_request_1! \vee B_0\_read\_request_2! \vee B_0\_read\_request_3!$

i.e., if A_2 receives a read request of level 1 and then sends a read request to B_0, then the unique next visible answer has level -1.

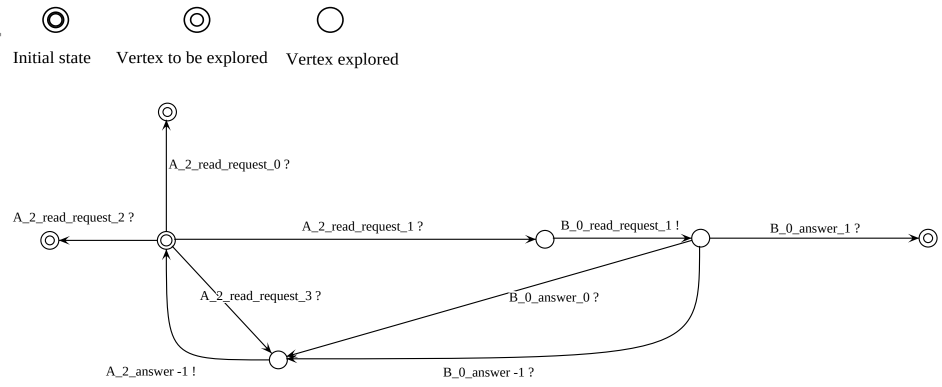

Figure 6: A_2 behaviour in the case of nested invocations.

4.2.2 Model–checking results and generalization

Formula F3 has been proven true on the model of $\mathtt{A}_2 \,||\, \mathtt{B}_0$. The LTS produced for $\mathtt{A}_2 \,||\, \mathtt{B}_0$ consists of 851 transitions and 280 states.

The generalization step deals with the integrity levels. We would consider all the models built with A_i and B_k, and $read\_request_j$, for any i, j, k such that $i \geq j > k$ and check the corresponding instances of the following general formula expressing Prop 3:

$$\mathop{\&}_{j \,\leq\, i} AG\,[A_i\_read\_request_j?]\; AG\,[\bigvee_{\substack{k\,<\,j\\ m\,=\,0\,...\,3}} B_k\_read\_request_m!\,]$$
$$A\,[true\{\sim \bigvee_{m\,=\,0}^{3} A_i\_answer_m!\}\; U\; \{A_i\_answer_{-1}!\}true]$$

We can repeat the model–checking for any i, j, k, and it is easy to see that the result of model–checking will not change. The proven properties guarantee that data do not flow from a given level of integrity to a higher level of integrity through a pair of nested invocations.

A further generalization step can be made, to conclude that data do not flow from a given level of integrity to a higher one, through any number of nested invocations which include an invocation to an object with a lower level than the level of initial read invocation (Prop G–3). We reason by induction. We consider n objects $\mathtt{B}^1 \,||\, \mathtt{B}^2 \,||\, \ldots \,||\, \mathtt{B}^{\mathtt{n}}$, where $\mathtt{B}^1$ is the first object of the chain, i.e. the one receiving the first read invocation. Let `j` be the integrity level of such an invocation. The inductive assumption guarantees that if any of the `B`s has an integrity level lower that `j`, then the answer has level `-1`. Thus, we can safely simulate the behaviour of the parallel composition $\mathtt{B}^1 \,||\, \mathtt{B}^2 \,||\, \ldots \,||\, \mathtt{B}^{\mathtt{n}}$ with an MLO $\mathtt{C}_{\mathtt{k}}$ with `k<j`. Hence, once we have proved that $\mathtt{A} \,||\, \mathtt{C}_{\mathtt{k}}$ behaves correctly, we can conclude that this is also true for $\mathtt{A} \,||\, \mathtt{B}^1 \,||\, \mathtt{B}^2 \,||\, \ldots \,||\, \mathtt{B}^{\mathtt{n}}$.

4.3 Concurrent invocations

We now consider the case of two concurrent invocations of the same object. This means that two requests sent to an object are served concurrently. The Multiple Levels of Integrity policy imposes the creation of two instances of the object to serve these two requests.

We model this through the definition of two concurrent copies of the object, each accepting one of the requests. Alternatively, we could have explicitly modelled object creation.

We chose the first option, according to what we have done in Section 3, where we modelled MLOs recursively, requiring the MLOs to restore their integrity level after answering a request. This way we could both satisfy the policy constraint of having an MLO with its intrinsic integrity level each time it is invoked and of having a finite state model associated with each system. On the contrary, explicit modelling of object creation generally leads to infinite state models.

Assume $\mathtt{A}_2$ is an MLO with *maxil* 2. We analyze the scenario corresponding to to the parallel composition $\mathtt{A}_2 \parallel \mathtt{A}_2$ when a read request of level 1 and then a write request of level 0 are received, and show that the write request does not influence the answer to the read request.

1. at the beginning we have $\mathtt{A}_2 \parallel \mathtt{A}_2$;
2. a $\mathtt{read\_request}_1$ is received;
3. the request is accepted, since $1 \leq maxil(\mathtt{A}_2)$:
4. we now have $\mathtt{SAT\_R\_REQ}_{1,2,2} \parallel \mathtt{A}_2$;
5. a $\mathtt{write\_request}_0$ is received;
6. the request is accepted unconditionally;
7. we then have: $\mathtt{SAT\_R\_REQ}_{1,2,2} \parallel \mathtt{SAT\_W\_REQ}_{0,0,2}$;
8. $\mathtt{answer}_2$ is sent back;
9. we are in situation $\mathtt{A}_2 \parallel \mathtt{SAT\_W\_REQ}_{0,0,2}$;
10. the computation started by the write request ends, and we are back to the initial situation $\mathtt{A}_2 \parallel \mathtt{A}_2$.

Fig. 7 gives a partial view of the global LTS representing the behaviour of $\mathtt{A}_2 \parallel \mathtt{A}_2$, with respect to the above scenario. To avoid multiple synchronizations, we have used (although not shown for simplicity) the CCS-MEIJE relabelling operator, and introduced the labels `first` and `second` to distinguish the actions of the two objects. We assume that each of the two copies of object $\mathtt{A}_2$ sends an ack after receiving a request. The ack to a read request can be $\mathtt{firstA_2\_called\_read\ !}$ or $\mathtt{secondA_2\_called\_read\ !}$, depending on which copy of $\mathtt{A}_2$ accepted the request.

We can observe from Figure 7, that the two invocations are interleaved, but independent, due to the stateless nature of the MLOs. Actually, this formalization has allowed the nature of concurrency of MLOs to be singled out. Given this independence, properties about non–interference are trivially guaranteed. Nevertheless, we report in the following the application of model checking verification to this case.

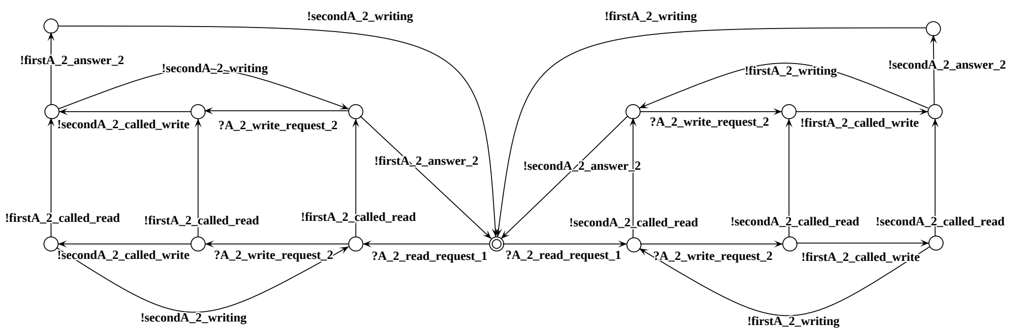

Figure 7: Concurrent Invocations.

4.3.1 Properties in the case of concurrent invocations

Prop 4 for the case of $\mathtt{A}_2$ || $\mathtt{A}_2$ receiving a read request of level 1 and a write request of level 0 is formalized as follows:

F4: $AG\ [A_2\_read\_request_1?]\ A\ [true\ \{true\}\ U\ \{A_2\_answer_2!\}\ true]$
&
$AG\ [A_2\_read\_request_1?]\ E\ [true\ \{\sim \mu\}\ U\ \{A_2\_write\_request_0?\}\ true]$

where μ stays for $A_2\_answer_{-1}! \vee A_2\_answer_0! \vee \ldots \vee A_2\_answer_3!$

This means that a write request of level 0, which decreases the MLO integrity level, can be accepted while a read request is being served, without influencing it: the two requests are dealt with independently by the two instances of $\mathtt{A}_2$.

4.3.2 Model–checking results and generalization

Formula F4 has been proven true on the model of $\mathtt{A}_2$ || $\mathtt{A}_2$. The LTS produced for validating the concurrent invocations case study consists of 1168 transitions and 400 states.

To generalize our result with respect to the integrity levels, we need to prove that the following formula holds for all A_is:

$$\underset{j<i}{\&}\ AG\ [A_i\_read\_request_j?]\ A\ [true\ \{true\}\ U\ \{A_i\_answer_i!\}\ true]$$

&

$$\underset{j<i}{\&}\ AG\ [A_i\_read\_request_j?]$$
$$E\ [true\ \{\sim \bigvee_{3}^{h=-1} A_i\_answer_h!\}\ U\ \{\bigvee_{k<j} A_i\_write\_request_k?\}\ true]$$

A simple solution is to repeat the proof for all the cases, using any A_i instead of A_2, $read\_request_j$ with any $j \leq i$ instead of $read\_request_1$, and $write\_request_k$ (for $k < i$) instead of $write\_request_0$, and prove that the model satisfies the corresponding instance of the above general formula expressing Prop 4.

Finally, according to the policy, any number of concurrent invocations of an object is allowed. For the sake of validation via model checking, we rather need to model the objects' behaviour via a finite state machine and thus we need to limit the number of possible concurrent invocations to a bound quantity. Of course, this can be as large as we want, thus obtaining a reasonable approximation of the general result.

A different solution, that does not need to bind the number of concurrent invocations, is to prove the effective independence between two concurrent MLO instances. To experiment this solution we proved by model checking that when one of the copies of $\mathtt{A_2}$ accepts a request, then it cannot accept other requests before answering the first one, as expressed by the following formula.

F5: $AG\ [firstA_2\_called\_read!]\ A\ [true\ \{\sim \mu\}\ U\ \{\bigvee_{y \leq 2} firstA_2\_answer_y!\}\ true]$

where

$\mu = firstA_2\_called\_read! \lor firstA_2\_called\_write! \lor firstA_2\_called\_read\_write!$

4.4 The state explosion problem

The application of the model checking approach can sometimes be affected by the so–called *state explosion* problem, when the parallel composition of several processes is considered. Indeed, when two or more concurrent processes are considered, the LTS dimensions can grow exponentially.

One way to solve the state explosion problem is to impose some restrictions on the behaviour of the system under consideration, to limit the number of states. These restrictions can be made by looking at the properties to be verified and cutting the paths which do not affect the validity of the property, if they exist.

Though the state explosion problem was not a concern for this validation case (see Table 4), we discuss how we could have restricted the behaviour of the nested invocation pattern of Section 4. The restriction of $\mathtt{A_2} \parallel \mathtt{B_0}$ could be operated in two steps: first constrain $\mathtt{A_2}$ to react to the request received by actually sending a read request to $\mathtt{B_0}$. This is the consequence of an assumption on the functional behaviour of $\mathtt{A_2}$. Let $\mathtt{A'_2}$ be the restricted $\mathtt{A_2}$. The second restriction involves the level of the request sent to $\mathtt{B_0}$. We prove that this level is 1, by checking the following formula against the model of $\mathtt{A'_2}$.

$AG\ [A_2\_read\_request_1?]\ A\ [true\ \{false\}\ U\ \{B_0\_read\_request_1!\}\ true]$

Formula	States	Transitions	Time
F1	23	96	$<$ 0,1 sec.
F2	23	96	$<$ 0,1 sec.
F3	280	851	$<$ 0,1 sec.
F3'	17	20	$<$ 0,1 sec.
F4	400	1168	$<$ 0,1 sec.
F5	400	1168	$<$ 0,1 sec.

Table 4: Summary of the model–checking results. Proofs were made on a Sun Ultra 1.

The restricted behaviour of $\mathtt{A}_2 \parallel \mathtt{B}_0$ is represented by the automaton fragment in Fig 6. The LTS has only 20 transitions and 17 states. Now, we can prove that the restricted model of $\mathtt{A}_2 \parallel \mathtt{B}_0$ satisfies the following formula:

F3': $AG\ [A_2\_read\_request_1?]\ A\ [true\ \{\mu\}\ U\ \{A_2\_answer_{-1}!\}\ true]$

which strengthens F3 asking that, when $\mathtt{A}_2$ receives a read request of level 1, then the unique next visible answer has level -1.

In the above reasoning, we have modified the processes by exploiting the process algebras restriction operator.

5 Related Work

The use of process algebras to validate interaction policies is not new. In [29], security policies are addressed. The authors introduce a new process algebra, called SPA, derived from CCS. In SPA, the set of actions is partitioned into two subsets corresponding to the two possible action levels: low and high. To verify security properties, they define the Compositional Security Checker, a tool that verifies the semantic equivalence between processes.

The main differences between their approach and ours lies in the action set partition, which we claim is not necessary. In particular, we can exploit standard process algebras and verification tools, while they have to define ad hoc ones. Moreover, we prefer to use a logical approach through model checking to the verification of interaction properties, since it is generally more abstract and also more efficient than the one based on behavioural equivalences.

Similar ideas are also used in the formalization and analysis of architectural styles in an operational framework [5]. The authors use a process algebra to formalize the interactional properties of components and connectors, abstracting from their functionalities. Similarly to us, they describe a component/connector with a term of a process algebra and interactions are specified through actions. They again use the notion of bisimulation equivalence to reason on the properties of the architecture, with particular interest in architectural compatibility and conformity.

An alternative approach to verify fault tolerant mechanisms is to exploit theorem proving techniques [30]. In the theorem proving approach, the system state is modeled in terms of set–theoretical structures, and operations are modeled by specifying their pre– and post–conditions in terms of the system state [12]. Properties are described by invariants that must be proved to hold through the system execution by means of a theorem prover, usually with the help of the user. This is a drawback with respect to model checkers, which are automatic tools. On the contrary, the proof theoretical approach can partially overcome some of the drawbacks of the model checking one, especially those related to state explosion problems.

Theorem proving and model checking are complementary techniques and some environments to combine them have recently been defined. One of the most remarkable is the integration of the BDD based model checker [14] for the propositional μ–calculus within the framework of the PVS proof checker [40, 42].

More recently, the STeP environment has been proposed [7, 8]. It supports a diagram based model checking procedure which can verify infinite state systems using STeP's deductive tools.

Abstraction from data and functional details, which maintains only behavioural information in a process algebraic style, has been recently adopted in the definition of the so called *behavioural type systems* [37, 36, 26]. These type systems associate to a component an abstraction of their behaviour in a suitable process algebra, aiming to check the compatibility of communicating concurrent objects, with regard to the matching of their respective behaviour.

The analogy with our approach lies in the interest to verification, though in the case of behavioural types the focus is in a verification of compatibility between components (which can be reduced to some form of equivalence or pre-order verification between two behaviours) while in our case the focus is on the verification of properties over a single behaviour, hence performed through model checking.

6 Discussion

A first point of discussion deals with the OO model to which the policy refers, which appear to be simple and static. We have received this model with the policy, as an input: the simplicity and staticity of the OO model adopted is related to the typical conservativeness of safety critical industries. For example, dynamic object creation is not recommended in the cited EN50128 norm due to unpredictability of memory exhaustion. Also, implicit invocations are not well accepted in a context in which the code is required to explicitly indicate data and control flows. Consequently, we have not addressed, in this work, the formalization of the implicit invocation interaction mechanism.

By the way, the conservativeness of the project context well matches the use of model checking, which requires a static configuration of interacting state machines. Nevertheless, we claim that a more general OO model can be dealt with within our approach.

In particular, we could address the process algebraic formalization of the implicit invocation mechanism as done, for instance, in [25]. Implicit invocation is an important architectural style for system design. The basic idea of this paradigm is that a component A can invoke a component B without knowing B's name. To model implicit invocation, the authors consider a collection of components that anonymously exchange messages (events) by means of a dispatcher and an event–component binding. A binding associates an event e with zero or more components that are to be triggered when the event is announced. Since CCS/MEIJE supports multi–way synchronization, it is possible to model systems exploiting implicit invocation. The CCS/MEIJE multi–way synchronization mechanism permits one to compose in parallel, on a multi–way synchronized channel, all the components of the system that should trigger when matching the event which is communicated on the channel. So, it is possible that zero or more components trigger.

A further point of discussion is why we have used CCS, even if many formalisms have been proposed after it. Our choice was guided by the following considerations:

- basic process algebras are suitable to model interaction policies, when functionalities of the object are abstracted away;
- the availability of a verification environment including several powerful verification tools - for example we have had a significant experience with LOTOS as well, and a few verification tools working on (basic) LOTOS are available, but in the end a verifiable LOTOS specification is not very different from our CCS one;
- the choice of CCS and JACK was made inside the GUARDS project, also to have a testbed on which to test and refine the verification tools themselves;
- CCS is anyway one of the formal methods listed in the EN50128 standard, so its use is potentially more easily accepted in an industrial context which looks compliance to that standard (one of the partners of the GUARDS project is indeed a railway signalling company);
- it is known that the plus operator is not suitable to model different aspects of non-determinism, and that it can be confusing and difficult to use. However, this is mainly a specification problem: when it comes to model-checking verification, where you need state machines, the CCS plus is exactly corresponding to the choice between alternate transitions. And

we have not met any difficulty with the plus in our modelling of the MLI policy;

- in a previous experience [4], we have not met strong opposition to CCS and ACTL modeling within an industrial context. Certainly should a more appealing formalism, with the same verification capabilities be available, we think that the approach could be fruitfully used. But again we want to insist that the use of a basic formalisms, directly related to a finite state representation, is in line with the conservativeness of the safety critical software industries.

- the formal specification and verification of the same policy has been made again by people in our group using the more popular PROMELA language and SPIN model-checker[32, 33]. This experiment, which followed the project outcomes, confirmed our results, but with no significant advantage in terms of easy of modeling and verification effort;

7 Conclusions

We have presented the application of a model–checking approach to specify and validate the Multiple Levels of Integrity protocol of GUARDS. A formal description of the protocol has been provided, in CCS/MEIJE style, and a set of properties have been formally stated in the ACTL temporal logic. Property satisfaction has been proved by exploiting the AMC model checker, available within the JACK verification environment.

The validation approach we have followed for the GUARDS Multiple Levels of Integrity policy can be exported to other contexts where specific properties of Object–Oriented distributed systems have to be checked. For example, a case which has much in common with the Multiple Levels of Integrity policy is the verification of security properties within distributed systems, in particular within Object–Oriented distributed systems. The high diffusion of Object–Oriented distributed infrastructures and frameworks (CORBA, TINA, ODP) makes this issue a hot topic where our approach to verification may prove useful. The verification of the "interactional" aspects of such distributed infrastructures can follow our approach.

Indeed, since interaction policies abstract from data and from functions, the description of a system adopting a policy can be reduced to the description of the possible interactions (method invocations) between the objects that make up the system.

We recall the validation path we have followed, in order to draw some guidelines for the application of the model–checking approach to the early verification of protocol scenarios other than the one considered.

1. abstraction from the number of instances of objects, which are often potentially infinite: we consider only the minimum number that is needed to analyse the interaction protocol considered.

2. modeling of the interaction protocol by process-algebra terms: we have singled out the events causing relevant state changes with respect to the policy. In our case, we have considered read and write method invocations.

3. definition of the desired properties by temporal logic formulae.

4. model–checking.

The model checking results have to be generalized by suitable reasoning: this step concerns the proof that generalization/concretization maintains the verified properties on the abstract model. The proofs presented in Section 4 were made without any automatic support. However, this is an issue where theorem proving tools can prove useful, and is a move in the direction of a stronger integration between model checking and theorem proving approaches.

References

[1] D.E. Bell and L.J. LaPadula. Security Computer Systems: Mathematical foundations and model. Technical Report Technical Report M74-244, MITRE Corp., Bedford, Mass., 1974.

[2] C. Bernardeschi, A.Fantechi, and S.Gnesi. Formal Validation of the GUARDS Inter–consistency Mechanism. *Reliability, Engineering and System Safety(RE&SS)*, 71(3):261–270, Feb. 2001. Elsevier.

[3] C. Bernardeschi, A. Fantechi, and S. Gnesi. Formal verification. Chapt. 8 of [38].

[4] C. Bernardeschi, A. Fantechi, and S. Gnesi. An Industrial Application for the JACK Environment. *Journal of Systems and Software*, 39(2), 1997. Elsevier Science Inc.

[5] M. Bernardo, P. Ciancarini, and L. Donatiello. On the Formalization of Architectural Types with Process Algebras. In *Proc. ACM SIGSOFT 8th Int. Symp. on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pages 140–148. ACM Press, 2000.

[6] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report Tech. Rep. ESD-TR 76-372, MITRE Co., Apr. 1997.

[7] N. Bjørner, Z. Manna, H. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253(1):27–60, 2001.

[8] Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[9] A. Bouali, S. Gnesi, and S. Larosa. JACK: Just another concurrency kit. *Bulletin of the European Association for Theoretical Computer Science*, 54:207–224, 1994.

[10] G. Boudol. Notes on Algebraic Calculi of Processes. NATO ASI Series F13, 1985.

[11] J.P. Bowen and M.G Hinchey. Seven more myths of formal methods. *IEEE Software*, 12:34–41, July 1995.

[12] R.S. Boyer and J.S. Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[13] G. Bruns and I. Sutherland. Model Checking and Fault Tolerance. In *Proc. 6-th International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 45–59, Sydney, Australia, 1997. Springer-Verlag.

[14] J.R. Burch, E.M.Clarke, K.L. McMillan, D. Dill, and J. Hwang. Symbolic Model Checking 10^{20} states and beyond. In *Proceedings of Symposium on Logics in Computer Science*, 1990.

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

[16] CENELEC. Railway Applications: Software for Railway Control and Protection Systems. CENELEC draft CLC/SC9XA/WG1 (sec), feb 1994.

[17] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An Open-Source Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[18] D.D Clark and D.R Wilson. Comparison of Commercial and Military Computer Security Policies. In *IEEE Symp. on Security and Privacy*, pages 184–194, Oakland, CA, 1987. IEEE Computer Society Press.

[19] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specification. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[20] E.M. Clarke, O. Grumberg, and D.Peled. *Model Checking.* MIT Press, 1999.

[21] E.M. Clarke and J.M. Wing. Formal methods: state of the Art and Future Directions. *ACM Computing Surveys*, 28(4):627–643, 1996.

[22] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action–based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks & ISDN Systems*, 25(7), 1993.

[23] R. De Nicola and F.W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[24] D.L. Dill, A.J. Drexler, A.J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[25] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 209–221. ACM Press, 1998.

[26] A. Nimour E. Najm and J.-B. Stefani. Garanteeing liveness in an object calculus through behavioral typing. In *Proceedings of FORTE/PSTV'99*, Beijing, China, October 1999. Kluwer.

[27] E.A. Emerson and J.Y. Halpern. Sometimes and Not Never Revisited: on Branching Time versus Linear Time Temporal Logic. *Journal of ACM*, 33(1):151–178, Jan. 1986.

[28] A. Fantechi, S. Gnesi, and L. Semini. Formal Description and Validation for an Integrity Policy Supporting Multiple Levels of Criticality. In C.B. Weinstock and J. Rushby, editors, *Dependable Computing for Critical Applications, 7*, pages 129–146. IEEE Computer Society Press, 1999.

[29] R. Focardi and R. Gorrieri. The Compositional Security Checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, September 1997.

[30] L. Gong, P. Lincoln, and J. Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults. In *Proc. DCCA–5, Fifth IFIP International Conference on Dependable Computing for Critical Applications*, Urbana-Champaign, Il, USA, 1995.

[31] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall Int., 1985.

[32] G.J. Holzmann. The Model Checker SPIN. *IEEE Transaction on Software Engineering*, 5(23):279–295, 1997.

[33] G.J. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004.

[34] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst.*, 21(1):46–89, 1999.

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

[36] B.C. Pierce N. Kobayashi and D.N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[37] O. Nierstrasz. *Regular types for active object*, pages 99–121. Prentice-Hall, 1995.

[38] D. Powell, editor. *A Generic Fault–Tolerant Architecture for Real–Time Dependable Systems.* Kluwer Academic Publishers, Jan. 2001.

[39] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings. GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[40] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 1995.

[41] RTCA/EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. RTCA–DO178B / EUROCAE ED–12B, 1992.

[42] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 1–16, State College, PA, August 2000. Springer-Verlag.

[43] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *To be presented at DSN '04*, Florence, June 2004.

[44] E. Totel, L. Beus-Dukic, J.-P. Blanquart, Y. Deswarte, V. Nicomet, D. Powell, and A. Wellings. Multi level integrity mechanism. Chapt. 6 of [38].

[45] E. Totel, J.-P. Blanquart, Y. Deswarte, and D. Powell. Supporting Multiple Levels of Criticality. In *Proceedings 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, Munich, Germany, Jun. 1998. IEEE Computer Society Press.