Using a Logical and Categorical Approach for the Validation of Fault-Tolerant Systems

C. SEGUIN, V. WIELS
{seguin, wiels}@cert.fr

CERT-ONERA, 2 av. E. Belin, BP 4025, 31055 Toulouse cedex, France

Abstract. We propose a categorical and logical formalism and apply it in order to compositionally specify and verify the fault-tolerance mechanisms of the Modulor system. We claim that our approach is well-suited to the validation of real-sized critical systems.

1 Introduction

In this paper, we present how we have formally validated some fault tolerant mechanisms of the Modulor system, using a combination of a logical and a categorical approach. Through this experiment, we aim to demonstrate the interest of the approach for validating critical real-sized systems.

The Modulor system was developed at CERT jointly with the design of a flexible, massively parallel machine architecture. It supplies the programmers with ad-hoc software tools allowing to take advantage of the processor net of the machine. We were interested in validating the principle of the detection mechanism dedicated to the machine. The validation process was mainly influenced by the following system features.

- First, the whole system is *distributed and large* because it is composed of numerous replications of a basic set of processes; each set of processes runs on a processor and interacts with other sets of processes by communication channels building a net of processors. A *modular* approach allows control of the complexity of the validation induced by the size of the system.
- Next, the topology of the net of processors is *flexible* and changes during a runtime. In order to have general results, the validation results must be *generic*, i.e. independent of the net topology.
- Finally, the system provides *fault-tolerance* mechanisms. Consequently, the validation process must deal with *fault modeling.*

Before using the validation approach described in this paper, we have carried out validations based on CCS and model-checking principles (see [13]) and exhibited limits of these approaches for our purpose.

On one hand, logical formalisms such as TLA ([10]) or Unity ([2]) are expressive enough to easily specify distributed systems and their properties. However, these formalisms lack means of structuring specifications and fully automatic verifying tools. These gaps make the formal validation of a real-sized system difficult.

On the other hand, algebraic approaches such as CCS ([12]) or CSP([8]) offer composition and communication laws allowing structuring of specifications. Moreover, a lot of analysis tools are currently available (bisimulation analysis, model-checking of μ-calculus formulae,...). However the operational way of modeling systems makes the fault modeling cumbersome in our case. Each faulty state must be pointed out individually by performing a "fault-action" which triggers other actions describing the ad-hoc erroneous behavior. The specification is more concise using a descriptive approach since the features of the faulty states can be summed up by a few generic formulae once and for all.

The lack of means of expressing state properties also penalizes the description of the data properties. In case of fault tolerant systems, redundancy resources must be taken into account. In the case of the Modulor system, these resources evolve with time and we must deal with their generic properties. Operational approaches are not well-suited to express state invariants.

Our proposal aims to bridge the gap between the two trends: we wish to keep the expressiveness of the logical formalisms and the composition laws a la CCS. We have based our work on the categorical framework proposed by Fiadeiro and Maibaum [6]; each component of the system is described by a logical theory and the whole system results from the interconnection of the components by means of categories. We have defined a new logic to describe a component, mixing linear temporal logic ([11]), and dynamic logic ([7]). The linear temporal logic has proved to be well suited for expressing properties of concurrent and reactive systems; and the behavior of systems is easily described thanks to the explicit use of actions in the dynamic logic. Our logic is, in fact, a fragment of a particular μ-calculus [9] , but we have found an axiomatic sound and complete with respect to the proposed semantics [17]. This result makes the proofs easier on a component theory. The properties of the global system can be derived in a modular way in the categorical framework.

In section 2, we present the Modulor machine and its fault-tolerance mechanisms. The aim of section 3 is to introduce our formalism, using as a simple example the synchronization at the beginning of a phase. Section 4 finally gives the results we obtained about Modulor and shows the adequacy of the formalism for the specification and verification of fault-tolerant systems.

2 Modulor

The research project named Modulor[1] is directed towards the design and realization of a massively parallel, modular and dynamically reconfigurable computer, as well as the design and realization of the associated software tools. The architecture is a network of processors, and the physical reconfigurability of the hardware links connecting the processors allows us to satisfy two objectives:

[1] Modulor project has been mainly funded by DRET (Research Division of the French Defense Department).

- An optimal topology can be chosen for each application or each phase of the application. We limited our objectives to explicit, quasi-dynamic reconfiguration. We ask programmers to decompose their application into algorithmic phases. Each phase is a graph of communicating processes, written up to now in Occam, and necessitates a specific communication topology.
- Reconfiguration capabilities can be used to offer the dynamic redundancy needed for fault-tolerant computing (the likelihood of the occurrence of a faulty processor grows with the number of such processors).

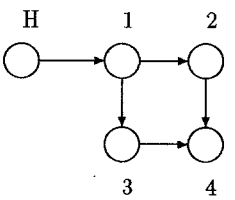
Up to now, the interest of the approach has been validated by implementing algorithms ranging from numerical algorithms through to tree searching algorithms. More details about these experiments can be found in [4].

The development of a reconfigurable application is assisted by a set of software tools. A lot of them have been validated by practical experiments. Now, our main concern is to formally validate :

- the synchronization mechanisms for the phase initialization and completion.
- the fault-tolerance mechanisms.

2.1 Features of the net processors

We call "net processors" the processors required by an application. All the net processors are connected by oriented links. Considering one link, we call the source processor "father processor" and the target one "son processor". Among the net processors, we distinguish the host processor. It is connected to one of the net processors and is not supposed to break down. In this paper, we will base all our illustrations on the following configuration:



The process running on the host processor plays a supervisory role with respect to the execution of the computation phase: it chooses the type of the following phase in accordance with the results of the previous one and synchronizes the beginning and the end of each phase.

2.2 Synchronization mechanisms

A round of synchronization message exchanges occurs at the phase initialization. The used algorithm allows each processor [2] to receive the phase number, and then to begin the algorithmic phase with the suitable topology.

[2] For the sake of simplicity, we will identify a processor with the process that it is executing.

During the synchronization, the host processor sends the phase number to its son and it begins the execution of the phase. A net processor waits for a phase number from one of its fathers. Then it sends it to all its sons and waits for it from all its other fathers (if it has any) ; it can send and receive in parallel. Finally, when all the sendings and receivings are finished, it begins the execution of the phase.

A second rendez-vous takes place at the phase completion, which is the completion of each process. A communication tree exists for this synchronization (the root of this tree is the host processor). When a process has ended its computation, it waits for all its sons to send an *OK* message and then sends an *OK* message to its father.
After this ending synchronization, the supervisor physically modifies the communication links for the next phase. At this time, no process of the network must send messages, it is only waiting for a new phase to begin. All these mechanisms are only written with message exchanges.

Synchronization properties. The synchronization algorithm aims at triggering the computation of the phase and ensuring the consistent use of the exchanged data during the phase. Consistency is reached when a process does not receive synchronization messages while it is waiting for data messages.

So in the context of our application, we may express more precisely these requirements by:

- P1: each process receives at least one synchronization message before beginning the phase computation;
- P2: each process receives no more synchronization messages after entering the phase computation.

2.3 Fault-tolerance mechanisms

Fault tolerance is achieved by passive redundancy. So detection, isolation and recovery mechanisms have been implemented (fault detection, diagnosis and reconfiguration of the architecture in order to replay the phase). We detail the detection mechanism because it is this part we are interested in for the validation.

We assume that *a failure of the processor or of one of the communication links corresponds to a total stop of the processor.* Besides, every processor attempting to communicate with a faulty processor is blocked. We also assume the user's code has been already validated and then *we assume a maximum time for each phase execution.* So exceeding this limit is assimilated with an error, and means that a processor is faulty. The added mechanisms allow the deadlock to be detected when timers exceed the upper limit for the phase.

On each net processor of the net, the running process is divided into three processes (timer, server and phase) in order to control the state of the phase during its execution.

- The phase process is responsible for the execution of the phase. To achieve this execution, it communicates with the processors it is connected with. The phase process can break down at any time during the execution, or it can become blocked while trying to communicate with a processor that has broken down. The phase process regularly asks the server the current state of the phase. If the state is not correct, it stops its execution. If it can finish its execution correctly, it signals the correct end to the server.
  The server process updates the state of the phase (which becomes in error if a time-out has happened). The server also gives the state of the phase to the phase process. Finally, it stops the timer when the phase has ended correctly.
- The timer process initializes a timer at the beginning of the phase. When the execution of the phase is not finished in time (the process is locked, waiting for a communication with a locked or faulty processor), a time-out happens and the timer process unlocks the phase process if necessary. Otherwise, the timer process is stopped by the server process.

Detection Properties. We focus our experiment on the detection algorithm. We stated at least four requirements which express the correctness of this algorithm:

- P3: Every failure is detected: "Each time a processor is faulty, the host knows it".
- P4: The detection allows to resume communications between non-faulty processors: "All blocked processors will be released".

3 Specifying a component by means of the logic of actions and time

We have designed a logic to deal in the same formalism with the specification of a system behavior and the expected properties. This logic inherits the way of specifying the system actions from the dynamic logic and the expression of the properties from linear temporal logic. We first present the logic and then illustrate how to use it with the synchronization algorithm at the beginning of a phase.

3.1 Logic of actions and time

Syntax. For a given component of a system, let $\Delta_C = (CONST_C, ATT_C, ACT_C)$ be the component signature. $CONST_C$, ATT_C, and ACT_C are three disjoint and finite sets of name symbols denoting respectively the name of constants, attributes and actions of the component. We introduce the new action symbol τ which does not belong to ACT_C and denotes external actions of other components outside C.

From Δ_C, we inductively construct the set of the well-formed formulae relative to the component C:

- if $x \in ATT_C$ and $c \in CONST_C$, $x = c$ is a wff for C
- if A and B are wff for C and $\alpha \in ACT_C \cup \{\tau\}$ then $\neg A$, $A \rightarrow B$, $A \wedge B$, $A \vee B$, $A \leftrightarrow B$, T, F, $[\alpha]A$, $< \alpha > A$, OA, $\Box A$, $\Diamond A$, $A\,Until\,B$ are wff for C.

$\neg$, $\rightarrow$, $\wedge$, $\vee$, $\leftrightarrow$ are the connectives of the classical logic whereas $[\alpha]$, $< \alpha >$, O, $\Box$, $\Diamond$, *Until* are modal operators which have the following meaning :
$< \alpha > A$: it is possible to execute α reaching a situation in which A is true.
$[\alpha]A$: every execution of α leads to a situation in which A is true.
OA :at the next step of the execution, A will be true.
$\Box A$: now and in the future, A will always be true.
$\Diamond A$: A will be true.
$A\,Until\,B$: A will remain true until B becomes true.

We will give the semantics and an axiomatic for the connectives $\neg$, $\rightarrow$, the family of modal operators $[\alpha]$ and the temporal operators $\Box$, O and *Until*. The meaning of the other connectives, modal operators and the formulae T (true) and F (false) is given by the definitions above:

$T \equiv_{def} A \rightarrow A$ $\qquad$ $A \leftrightarrow B \equiv_{def} (A \rightarrow B) \wedge (B \rightarrow A)$
$F \equiv_{def} \neg T$ $\qquad$ $< \alpha > A \equiv_{def} \neg[\alpha]\neg A$
$A \wedge B \equiv_{def} \neg(A \rightarrow \neg B)$ $\qquad$ $OA \equiv_{def} \bigvee_{\alpha \in ACT_C \cup \{\tau\}} < \alpha > A$
$A \vee B \equiv_{def} \neg A \rightarrow B$

Semantics The formulae of our logic are interpreted within an infinite sequence of states standing for a computation of a component. In order to deal explicitly with actions, we label the transitions from one state to its immediate successor in the sequence, by a subset of $ACT_C \cup \{\tau\}$. We do not assume any hypothesis about the parallelism but an interleaving semantics can be gained by labeling the transition by an unique name of $ACT_C \cup \{\tau\}$.

We chose a linear time semantics rather than a branching one as proposed by Fiadeiro and Maibaum, because it seems to be expressive enough for our purpose without requiring the complexity of a temporal logic such as CTL* [5] .

Model definition. A model M_C relative to the signature $\Delta_C = (CONST_C, ATT_C, ACT_C)$ of a component is a quintuplet $(W, \{R_\alpha | \alpha \in ACT_C \cup \{\tau\}\}, \Pi, D, I_{CONSTC}, I_{ATTC})$ where:

- W is a non empty set of worlds (or states)
- each R_α is a partial function over W. R_α(w)=w' also noted by wR_αw' means that the successor w' of w is reached by performing α.
- Π is the next-state function which associates with each world its successor in the computation. In order to relate time and action, we add the following constraint: $\Pi = \bigcup_{\alpha \in (ACT_C \cup \{\tau\})} R_\alpha$, which expresses that a transition always results from performing at least one action.
- D is a domain of values
- I_{CONSTC} is the function of interpretation of the constants. The interpretation of the constants is time independent. $I_{CONSTC} : CONST_C \rightarrow D$

- I_{ATTC} is the function of interpretation of the attributes. This function is time dependent since the value of an attribute may change from one state to the other. $I_{ATTC} : ATT_C \times W \rightarrow D$

Satisfiability relation. The satisfiability relation $\models$ between pairs (M_C, w) and formulae is defined by the following rules :
$(M_C, w) \models (x = c)$ iff $I_{ATTC}(w, x) = I_{CONSTC}(c)$
$(M_C, w) \models \neg A$ iff $not((M_C, w) \models A)$
$(M_C, w) \models A \rightarrow B$ iff $(M_C, w) \models \neg A$ or $(M_C, w) \models B$
$(M_C, w) \models [\alpha]A$ iff $\forall w' \in W$, if $wR_\alpha w'$ then $(M_C, w') \models A$
$(M_C, w) \models OA$ iff $(M_C, \Pi(w)) \models A$ [3]
$(M_C, w) \models \Box A$ iff $\forall i \geq 0, (M_C, \Pi^i(w)) \models A$
$(M_C, w) \models A\, Until\, B$ iff $\exists i \geq 0$ such that
$(M_C, \Pi^i(w)) \models B$ and $\forall j < i, (M_C, \Pi^j(w)) \models A$

Satisfiability. A formula A is satisfiable if there is a model M and a world w of the set of world of M such that $(M, w) \models A$.
Validity. A formula is valid iff it is satisfiable in every world of every model.

Axiomatic The following system captures the semantics given in the previous section.

- Axioms of the propositional calculus and of the equality predicate
- Axiom of the propositional dynamic logic : $[\alpha](A \rightarrow B) \wedge [\alpha]A \rightarrow [\alpha]B$
  We recall the linkage definition between action and time:
  $OA \equiv_{def} \bigvee_{\alpha \in ACTC \cup \{\tau\}} < \alpha > A$
- Axioms of the propositional linear temporal logic :

$\Box(A \rightarrow B) \wedge \Box A \rightarrow \Box B$	$OA \leftrightarrow \neg O \neg A$
$\Box(A \rightarrow OA) \wedge A \rightarrow \Box A$	$A\, Until\, B \rightarrow \Diamond B$
$\Box A \rightarrow A \wedge O\Box A$	$A\, Until\, B \leftrightarrow B \vee (A \wedge O(A\, Until\, B))$

  Inference rules:

- R1 $\frac{\vdash A, \vdash A \rightarrow B}{\vdash B}$ (modus ponens)
- R2 $\frac{\vdash A}{\vdash \Box A}$ ($\Box$ necessitation)
- R3 $\frac{\vdash A}{\vdash [\alpha]A}$ ($[\alpha]$ necessitation)

Theorem. We have proved that the restriction of the proposed semantics to the pure propositional models is sound and complete with respect to the axiomatic below [17].

[3] As OA is defined by a combination of operators $< \alpha >$, the definitions of the relation Π and of the satisfiability relation for OA are not required; however we gave them for sake of readability.

3.2 Specification of a component

Each component C of a system can be described by a theory Φ_C of the logic of action and time presented above, using the object-oriented methodology proposed by Fiadeiro and Maibaum. The key point is the encapsulation principle allowing compositional specification and validation.

The component encapsulation

Principle. A component is defined by attributes and specific actions. Attributes are the state variables of the component; their values range over a set of constants. The actions are the "methods" related to the component. The encapsulation principle for a component C is stated as follows:

- the attributes and the actions of a component are only those whose names occur in the signature Δ_C;
- the possible values of an attribute range over the constants denoted by the names of $CONST_C$.
- Moreover, the scope of the effects of an action are local to the component; so, the attribute values of a component C can only be modified by the actions of ACT_C.

Indeed, the signature Δ_C is a way to circumscribe a component.

Modeling encapsulation in a theory Φ_C. To take into account the locality of the action effects, we add a locality axiom in each theory describing a given component. For a given component C and its signature $\Delta_C = (CONST_C, ATT_C, ACT_C)$, the locality axiom has the following pattern :

$$locusC : \bigvee_{\alpha \in ACTC} < \alpha > True \vee (\bigwedge_{x \in ATTC} (\bigvee_{c \in CONSTC} (x = c \wedge O(x = c))))$$

This axiom means that either an action of the component C is performed or the attributes keep their values in the next state.

Semantics point of view. Each time a transition is labeled by τ (i.e. by no action of the observed component), then the values of the component attributes remain unchanged after the transition.

According to the Fiadeiro and Maibaum's terminology, we call a model $M_C = (W, \{R_\alpha | \alpha \in ACT_C \cup \{\tau\}\}, \Pi, D, I_{CONSTC}, I_{ATTC})$ a Δ_C-locus with respect to the signature Δ_C iff:
$\forall (w, w') \in W$, if $wR_\tau w'$ and $R_\alpha(w)$ is undefined for all α of ACT_C,
then $\forall x \in ATT_C, I_{ATTC}(x, w) = I_{ATTC}(x, w')$

Let M_{locusC} be the set of models for C which are Δ_C-loci. We proved ([17]) that the locus axiom is exactly defined by the set of M_{locusC} :
$\forall M_C, \forall w, (M_C, w) \models locusC$ iff $M_C \in M_{locusC}$

We say a formula A is **loci-valid** and write $\models_{loci} A$ if
$\forall M_C \in M_{locusC}, (M_C, w) \models A$.

Description theory of a component We saw in section 2 that each computation phase on the Modulor machine begins with a synchronization of all the active processes of the phase. We present here the logical theory describing the behavior of one processor during this synchronization round.

Signature. For the description of a process, **three attributes** are needed: *ready_to_send, connect_to_j* and *connect_from_j*. The attribute *ready_to_send* takes the value $True$ [4] when the process has received a number for the phase, it means it is now able to send it to its sons. The attributes *connect_to_j* and *connect_from_j* describe the connections that exist between the processors. They may take three values: 0 when the connection does not exist, 1 when the connection exists and 2 when the connection with the processor j exists and the number has been sent or received to or from the processor j.

The process can execute **three actions**: send the number to a process j *send_j*, receive the number from a process j *receive_j* and begin the phase *begin_phase*.

Theory. Let us now define the theory associated with a component. First we express the properties of the attributes. Their values range over subset of constants and cannot have simultaneously two different values. These properties are **integrity constraints** which must always be satisfied during a computation. In the case of the theory of a process in Modulor, we express this by the formula: $\Box(ready\_to\_send = True \oplus ready\_to\_send = False) \wedge \Box(connect\_from\_j = 0 \oplus connect\_from\_j = 1 \oplus connect\_from\_j = 2) \wedge (connect\_to\_j = 0 \oplus connect\_to\_j = 1 \oplus connect\_to\_j = 2)$ where $\oplus$ stands for the exclusive or.

In order to reason with initialized computation, we may add formulae describing the **initial state of the attributes**. For instance, we have:
$ready\_to\_send = False$

Then we define formulae to express the features of the actions. We determined four classes of formulae:

- **necessary preconditions** to perform an action; for example, the process can only receive the number from a process j if j is connected with it:
  $\Box(< receive\_j > T \rightarrow connect\_from\_j = 1)$
- **reactivity or fairness formulae** to guarantee the trigerring of enable actions. For example, we have:
  $\Box(ready\_to\_send = True \wedge connect\_to\_j = 1 \rightarrow \Diamond < send\_j > T)$
  to ensure the action will be done if the preconditions are satisfied.
- **weakest post-conditions** to describe the effects of the actions. For example, after the reception of the number from j, the attribute *ready_to_send* takes the value True and *connect_from_j* the value 2:
  $\Box([receive\_j](connect\_from\_j = 2 \wedge ready\_to\_send = True)$
- **frame axioms** which characterize what attribute values remain unchanged after an action. These axioms may be seen as specialization of the locus

[4] The two values $True$ and T must be distinguished: T is a wff of our logic, $True$ denotes a constant.

axiom: they express that an attribute value changes during a transition if and only if some actions are performed. For instance:
$\Box(connect\_from\_j = 1 \wedge O(connect\_from\_j = 2) \rightarrow < receive\_j > T)$

The behavior of a process during the synchronization can then be summarized (forgetting the frame axioms, the integrity constraints and the initialization) as follows:

$Constants : True, False, 0, 1, 2$
$Attributes : ready\_to\_send, connect\_from\_j, connect\_to\_j$
$Actions : \quad send\_j, receive\_j, begin\_phase$
$Axioms : \quad \Box([receive\_j](connect\_from\_j = 2 \wedge ready\_to\_send = True)$
$\Box(< receive\_j > T \rightarrow connect\_from\_j = 1)$
$\Box([send\_j]connect\_to\_j = 2)$
$\Box(< send\_j > T \rightarrow connect\_to\_j = 1 \wedge ready\_to\_send = True)$
$\Box(ready\_to\_send = True \wedge connect\_to\_j = 1 \rightarrow \Diamond < send\_j > T)$
$\Box(< begin\_phase > T \rightarrow \bigwedge_{j \in [0..5]}((connect\_to\_j = 0 \vee connect\_to\_j = 2)$
$\wedge(connect\_from\_j = 0 \vee connect\_from\_j = 2)))$

4 Combining components by means of categories

In this section, we present the category of object descriptions defined by Fiadeiro and Maibaum in [6], and we consider the descriptions given in section 2 as objects of this category. Our aim in this section is to explain that the categorical framework offers a well-suited structure for the combination of components. In the category of descriptions, interactions between objects are described by the morphisms of the category, and we will see that the characteristics of these morphisms have some consequences on the types of composition which can be achieved between the descriptions. A diagram is obtained by linking the descriptions of all the components with the morphisms representing the relationships between them. This diagram can finally be collapsed in an object of the category whose formulae specify the behavior of the whole system.

4.1 The category of descriptions.

First, we recall the definition of a category. A category is composed of two collections: the *objects* of the category and the *morphisms* of the category, and of four operations. Two of these operations associate with each morphism f of the category respectively its domain $dom(f)$ and its codomain $cod(f)$, both of which are objects of the category. One writes $f : C \rightarrow D$ to indicate that f is a morphism with domain C and codomain D. The other two operations are an operation which associates with each object C of a category a morphism I_C called identity morphism and an operation of composition which associates to any pair (f, g) of morphisms such that $dom(f) = cod(g)$ another morphism $f \circ g$. These operations are required to satisfy the following axioms:

$dom(I_A) = A = cod(I_A)$

$dom(f \circ g) = dom(g)$, $cod(f \circ g) = cod(f)$
$I_A \circ f = f$, $f \circ I_A = f$
$(f \circ g) \circ h = f \circ (g \circ h)$

In this paper, we will consider the category of descriptions defined by Faideiro and Maibaum. The objects of this category are descriptions like the ones presented in section 2 (that is to say composed of a signature and of a set of formulae specifying the behavior of the processes described. We have changed the logic used to build the formulae but the structure of a description has been kept). The morphisms of this category are defined as follows:
Given object descriptions (Δ_1, Φ_1) and (Δ_2, Φ_2), a description morphism σ : $(\Delta_1, \Phi_1) \rightarrow (\Delta_2, \Phi_2)$ is a signature morphism $\Delta_1 \rightarrow \Delta_2$ such that:
$\models_{loci\Delta 2} \Phi_2 \rightarrow \sigma(F)$, for all axiom F of Φ_1
$\models_{loci\Delta 2} \Phi_2 \rightarrow \sigma(Locus_{(\Delta_1, \Phi_1)})$

where a signature morphism is defined by: given two object signatures $\Delta_1 = (CONST_1, ATTR_1, ACT_1)$ and $\Delta_2 = (CONST_2, ATTR_2, ACT_2)$, a signature morphism $\sigma : \Delta_1 \rightarrow \Delta_2$ consists of
for each c_1 in $CONST_1$ a constant symbol $\sigma(c_1)$ in $CONST_2$
for each a_1 in $ATTR_1$ an attribute symbol $\sigma(a_1)$ in $ATTR_2$
for each act_1 in ACT_1 an action symbol $\sigma(act_1)$ in ACT_2

From the definition of description morphisms, we know that the axioms of the source description are translated to theorems of the target description; but we can also infer the following property: given a morphism $\sigma : (\Delta_1, \Phi_1) \rightarrow (\Delta_2, \Phi_2)$, *If* $\models_{loci\Delta_1} \Phi_1 \rightarrow F$, *then* $\models_{loci\Delta 2} \Phi_2 \rightarrow \sigma(F)$

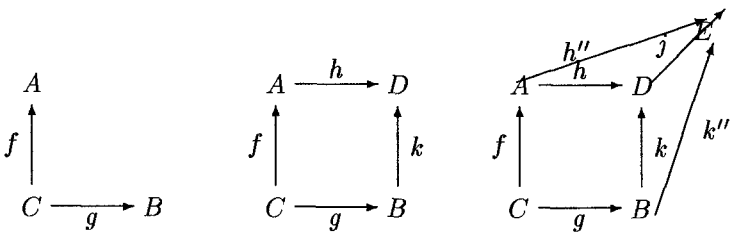
All the properties of a specification can thus be exported along a description morphism. Moreover, the axiom of locality is translated to a theorem of the target description.

4.2 Combining two components.

In the introduction, we said we would like to keep composition laws a la CCS. Particularly, as we are dealing with concurrent and reactive systems, parallel composition and communication mechanisms (like the synchronization of CCS) would be very useful. In this section, we will see how these combinations can be achieved with the categories. In the categorical framework, morphisms are the tools to use to express relationships between components. The general principle to describe an interaction between two components is to create a common subcomponent in which they synchronize. Given two objects A and B, we create an object C and two morphisms f and g such that $f : C \rightarrow A$ and $g : C \rightarrow B$. It can be represented by the first scheme of the following figure.

Then, to get an object describing the combination of A and B with correspondence on the elements of C, we build the push-out of this diagram. The push-out of such a diagram consists of another object D of the category together with two morphisms $h : A \rightarrow D$ and $k : B \rightarrow D$ such that, on the one hand, the second scheme of the following figure commutes (i.e. $h \circ f = k \circ g$, i.e. only one

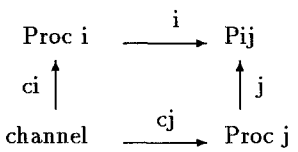
copy of C is obtained in D); and on the other hand, D is minimal, i.e. for every commuting diagram (cf last scheme of the following figure) there is a unique morphism $j : D \rightarrow E$ such that $j \circ h = h''$ and $j \circ k = k''$.



In the category of descriptions, the push-out corresponds to the parallel composition of two processes (A and B) with synchronization on the elements of the sub-component C. We are going to illustrate this method on the example of the composition of two processes of the Modulor machine.

Suppose we have the descriptions of two processes i et j and we want to describe the parallel composition of these 2 processes, taking into account that when Pi sends a message to Pj, this action corresponds, for Pj, to a reception of a message from Pi. It is sufficient to create a new object *channel* that contains one attribute and one action: *attr1* and *act1* and two morphisms between this object and Pi and Pj respectively.
Morphism ci: $attr1 \rightarrow connect\_to\_j, act1 \rightarrow send\_j$
Morphism cj: $attr1 \rightarrow connect\_from\_i, act1 \rightarrow receive\_i$
The role of the morphisms *ci* and *cj* is to indicate that *send_j* and *receive_i* must correspond to the same action in the object that will describe the parallel composition of the 2 processes (idem for the attributes).

When we compute the push-out of this diagram, we get the following diagram:



The resulting description Pij is the following, where *connectionij* corresponds to the *attr1* in *channel* and *communicationij* to the *act1* in channel and where the attributes and actions that take no part in the interaction between Pi and Pj stay the same but are prefixed by i or j in order to avoid conflicts of names. We do not give the axioms that are not concerned by the interaction: they are the same that in the example of 3.2, just prefixing the attributes and axioms by i or j (for example, $\Box[i.receive\_m]i.connect\_from\_m = 2$).

Constants : $True, False, 0, 1, 2$
Attributes : $connectionij, i.ready\_to\_send, j.ready\_to\_send$
$i.connect\_from\_m, j.connect\_to\_m$
$i.connect\_to\_m$ (m not equal to j), $j.connect\_from\_m(m \neq i)$
Actions : $communicationij, i.begin\_phase, j.begin\_phase$
$i.send\_m$ (m not equal to j), $j.receive\_m$ (m not equal to i)
$i.receive\_m, j.send\_m$
Axioms : $\Box[communicationij]connectionij = 2$
$\Box(i.ready\_to\_send \wedge connectionij = 1 \rightarrow$
$\Diamond < communicationij > T)$
$\Box[communicationij]j.ready\_to\_send = True$

4.3 Description of the global system.

In order to describe a system, we have seen that each component must be first described individually, then all these components must be interconnected with morphisms and sub-components. A diagram is then obtained. For instance, in Modulor, the diagram is composed of the five processes interconnected by sub-objects and morphisms. In the previous paragraph, we built an object representing the parallel composition of two components (push-out). Now, we want to obtain the parallel composition of all the components of the diagram, i.e. we have to generalize the notion of push-out for an arbitrary diagram. This notion exists: it is the colimit of a diagram.

Push-outs and more generally colimits can be calculated in a category if this category is finitely cocomplete. Fiadeiro and Maibaum have proved that the category of descriptions is finitely cocomplete, but more generally, there are at least two ways to build a finitely cocomplete category: the Comma construction and the finding of an initial object, coproducts and coequalizers; these two ways can be found in [15] and they have been implemented in a tool [16].

Let us see how to build this description concretely. We call 0,1,2,3,4 the morphisms between the descriptions of the processes (host,1,2,3,4) and the global description *system*. All the attributes and actions of the processes will be translated through these morphisms to become attributes and actions of the system. But the correspondences imposed by the sub-objects (like *channel*) must be taken into account. So there is no actions *send_i* or *receive_j* left but only *communicationji*, and the same thing for the connections. The axioms of the resulting theory are:
$\Box[communicationij]connectionij = 2$
$\Box(i.ready\_to\_send \wedge connectionij = 1 \rightarrow \Diamond < communicationij > T)$
$\Box[communicationij]j.ready\_to\_send = True$
$\Box(< i.begin\_phase > T \rightarrow \bigwedge_{j \in [0,5]}((connectionji = 0 \vee connectionji = 2) \wedge (connectionij = 0 \vee connectionji = 2)))$

Comment: the genericity of our formalism must be noticed here. With the previous method, we build a description of the system which is itself an object

of the category of descriptions and which may be reused as a component of an other diagram. We will see an example of this genericity in the next section.

5 Verification

In this section, we present how validation of a system specification can be achieved in the proposed hybrid framework. We first deal with the expression of the system properties and then describe our verification methodology.

5.1 Expressing properties.

By inheriting both from the linear temporal logic and the dynamic logic, our logic allows dual ways to express the properties of the system. If the stress is put on action, we may express properties in a mu-calculus fashion thanks to the predefined temporal operators instead of fix-point formulae and the actions operators (see e.g. the user manual of the Concurrency Workbench [3] for an outlook in the branching time case). And we can also express properties by means of state formulae combined by temporal operators (and this is often easier).

We give the following examples of properties for the synchronization algorithm.

- P1: each process receives at least one synchronization message before beginning the phase computation;
- P2: each process does not receive any more synchronization messages after entering the phase computation.

These properties can be expressed by the formulae:
$\Box(< begin\_phase > T \rightarrow ready\_to\_send = True)$
$\Box([begin\_phase] \bigwedge_{j \in [0,5]} (\Box \neg < receive\_j > T))$

5.2 Verification strategy

Our hybrid framework suggests a strategy to break down the complexity of the proofs. We based the strategy on exploiting the structure of the composed system.

- For a given global property, we scan the constraint, attribute and action names occuring in the formula. We intend to select the smallest component description which may enable the proof. The order between theories is induced by the description morphisms used to interconnect components: we say (Δ_2, Φ_2) is greater than (Δ_1, Φ_1) if the descriptions are connected in the whole system by a morphism f: $\Delta_1 \rightarrow \Delta_2$. If the heuristic succeeds, we can put back the theorem in the wished theory by means of the morphisms.
- When this heuristic fails, or when the least description results from colimit computation, we try to decompose the property in elementary lemmas provable in more basic descriptions (and these lemmas are translated along the morphisms to the global description).

- In the worst case, we have to prove the property in the object which describes the global system.

Comments on the automation:

- The development of a tool assisting the computation of push-outs and colimits of diagrams is experimented in CERT ([16]).
- We have proved that our logic is decidable ([17]); the sizes of the elementary component descriptions enable us to use automatic decision procedure which are always inefficient when the size of the specification is too big.
- The decomposition of properties in sub-properties requires a priori a good knowledge of the application. However, the encapsulation of the components and the frame axioms lead to a common strategy to prove liveness or "until" properties: we are looking for the actions which may modify the values of particular attributes. This strategy can be formalized in proof plans as in Bundy's approach in order to automatize the complete proof process [1]. Moreover, the proposed normalization of component descriptions should make the task easier. We show in the following section how these ideas may be exploited to lead proofs.

6 Results about Modulor

We saw in the previous section how to specify the synchronization at the beginning of a phase. We did not write down the verification of the properties, but we will show an example of proof in this section.
We are now concerned with the detection mechanism of Modulor (see the description of the fault-tolerance software tools in section 2). We want to prove that this mechanism is correct. We decompose this verification in two parts:

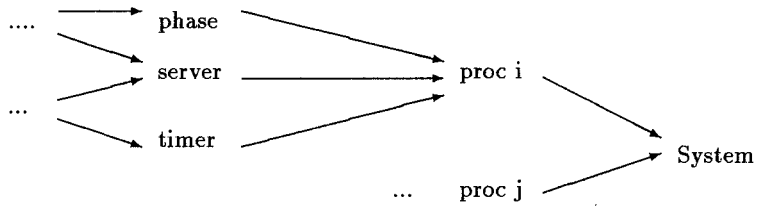
- the fact that, locally (on a processor), if the process is blocked, this will be detected and there exists a time where the process will be released (local detection)
- the fact that, if there is a faulty processor somewhere in the net, the host processor will be blocked (propagation due to the synchronization at the end of the phase).

With these two parts, we prove the correction of the detection. Indeed: if a fault occurs somewhere in the net, the host processor will be blocked (propagation), this will be detected and the host will be released (local detection), so it will be able to start a diagnostic phase. The other processors will be either OK and waiting for a new phase to begin, or faulty (and this will be found by the diagnostic phase), or blocked and in this case it will be detected and the processor will be released (local detection again).

6.1 Local detection

We will just present the specification of the local detection, because it is the most interesting part: we take advantage of the modularity and genericity of

the formalism for this specification. Indeed, in order to model this detection mechanism (see section 2) on a processor, we have to consider three processes: the phase process, the server and the timer. So we can consider each object *proci* as build from three smaller objects as shown by the following scheme:



So we have two levels: from the three processes (phase, server, timer), we build the description of a processor in Modulor. This description is itself an object that can be combined with other objects (the other descriptions of processors) to build a description of the system. So here, we have a good example of the genericity of our formalism.
For the description of these three processes, we try to be as close as possible to the real implementation. When the phase process ends correctly, it notices this correct end to the server that stops the timer. When a fault occurs, a time-out happens in the timer, the timer warns the server that changes the state of the phase and finally the timer releases the phase process.

Description of the phase process.
The phase process represents in fact the behavior of the process during a phase; it contains the synchronization at the beginning of the phase (described in previous section), the synchronization at the end of the phase (see after), and the behavior during the phase: the computation itself that will not be described and the additional actions that allows to detect a fault. It is this last part we are interested in.
We suppose we have an attribute *end_ok* which takes the value *True* when the phase has ended correctly. The part we are interested in is described by the following axioms:
$\Box < send\_server\_ok > T \rightarrow end\_ok = True$
$\Box[release]BEG = True$
$\Box(faulty = True \leftrightarrow \Box \bigwedge_{a \in ACT} \neg < a > T)$
where BEG is an attribute that takes the value $True$ when the process is ready to begin a new phase.

Description of the server.
$\Box[time - out]state = False$
$\Box[receive\_ok] < stop - timer > T$

Description of the timer
$\Box(\Box\neg < send\_server\_ok > T) \rightarrow \Diamond < time - out > T$
$\Box < time - out > T \rightarrow (\Box\neg < send\_server\_ok > T)$

$\Box[time - out] < release > T$
$\Box[stop - timer]end\_timer = True$

Comment: the property $\Box\neg < send\_server\_ok > T$ corresponds to the fact that the process is blocked (it will never be able to end correctly).

Then we have to build the description of a greater object that is the combination of the three with evident correspondences. And finally, we have to verify the following property which expresses the fact that if the process is blocked, it will be detected and the process will be released:
$(\Box\neg < send\_server\_ok > T) \rightarrow \Diamond(state = False \wedge BEG = True)$

6.2 Propagation-Synchronization at the end.

In this paragraph, we want to show that, if there is a faulty processor somewhere in the net, the host processor will be blocked at least during the synchronization at the end of the phase. In order to specify this synchronization at the end, we adopt the same methodology as for the synchronization at the beginning of the phase: we describe the behavior of a process and then we build the description of the system (cf example in 3.2). We do not give details because it is the same kind of description (it must just be noticed that the connections are not the same because here we have a tree (cf 2.2, synchronization mechanisms)). The description of a process is the following:

$Constants$: $True, False, 0, 1, 2$
$Attributes$: $aconnect\_to\_j, aconnect\_from\_j, faulty$
$Actions$: $send\_ok\_j, receive\_ok\_j$
$Axioms$: $\Box([receive\_ok\_j]aconnect\_from\_j = 2)$
$\Box(< receive\_ok\_j > T \rightarrow aconnect\_from\_j = 1)$
$\Box([send\_ok\_j]aconnect\_to\_j = 2)$
$\Box(< send\_ok\_j > T \rightarrow \bigwedge_{m\in[0,5]}(aconnect\_from\_m = 2 \vee$
$aconnect\_from\_m = 0) \wedge aconnect\_to\_j = 1$
$\Box(faulty = True \leftrightarrow (\Box \bigwedge_{a\in ACT} \neg < a > T))$

The last axiom shows how we model the fault: when a processor becomes faulty, it is not able to execute actions any more.

For the host, the description is the same, but an attribute *end_ok* must be added as well as the following axiom:
$\Box(end\_ok \leftrightarrow \bigwedge_{j\in[0,5]}(aconnect\_from\_j = 0 \vee aconnect\_from\_j = 2))$

Description of the system. We do not give the description of the system, but it can easily be obtained in the same way as in the previous section.

Expressing properties. The property we want to verify is:
P4: if one of the processor is faulty, the host process will be blocked.

This property can be expressed by the formula:
$\Box(1.faulty = True \vee 2.faulty = True \vee 3.faulty = True \vee 4.faulty = True) \rightarrow \Box 0.end\_ok = False$

Verification. We are going to verify the property in the case where this is the processor 4 which is faulty. We use intensively the modularity of our formalism. Indeed, the proof is decomposed in four lemmas and each lemma can be verified on a single process. These four lemmas are:

- if proc4 is faulty then it does not send any message *ok* to proc2. This lemma must be verified on proc4 and can be expressed in the following way:
  $faulty = True \rightarrow \Box\neg < send\_ok\_2 > T$
- if proc2 does not receive any message *ok* from proc4, then it does not send any message *ok* to proc1. This lemma must be verified on proc2 and can be expressed in the following way:
  $\Box\neg < receive\_ok\_4 > T \rightarrow \Box\neg < send\_ok\_1 > T$
- if proc1 does not receive any message *ok* from proc2, then it does not send any message *ok* to the host. This lemma must be verified on proc1 and can be expressed in the following way:
  $\Box\neg < receive\_ok\_2 > T \rightarrow \Box\neg < send\_ok\_0 > T$
- if the host does not receive any message *ok* from proc1, then *end_ok* will always stay false. This lemma must be verified on the host and can be expressed in the following way:
  $\Box\neg < receive\_ok\_1 > T \rightarrow \Box end\_ok = False$

The first step is given by the axiom:
$\Box(faulty = True \leftrightarrow (\Box \bigwedge_{a \in ACT} \neg < a > T))$

The second and third steps have the same demonstration, so we just give it for the second one. We work inside the description of proc2. We have:
$(\Box(< send\_ok\_j > T \rightarrow \bigwedge_{m \in [0,5]}(aconnect\_from\_m = 2 \vee aconnect\_from\_m = 0) \wedge aconnect\_to\_j = 1)$
For proc2, the only j for which $aconnect\_from\_j$ is equal to 1 is proc1 ; and the only m for which $aconnect\_from\_m$ is different from 0 is 4. So the precondition for proc2 sending a message *ok* to proc1 is that $aconnect\_from\_4 = 2$. The only means for $aconnect\_from\_4$ to take the value 2 is to receive a message *ok* from proc4:
$\Box([receive\_ok\_4]aconnect\_from\_4 = 2)$ So we have proved the second step.

There is the last step left. It can be proved inside the description of the host. A similar reasoning as for the second step can be used to state that the only mean for $aconnect\_from\_1$ to take the value 2 is to receive a message *ok* from proc1. So if we suppose that the host does not receive any message *ok* from proc1, we know that $aconnect\_from\_1$ will always keep the value 1 and so, because of:
$\Box(end\_ok \leftrightarrow \bigwedge_{j \in [0,5]}(aconnect\_from\_j = 0 \vee aconnect\_from\_j = 2))$
end_ok will always stay false.
To get the complete proof in the global description of the system, we then translate the 4 properties along the morphisms (4,2,1,0 respectively) into the object *system* ; and due to the correspondence of
$4.send\_ok\_2$ and $2.receive\_ok\_4$
$2.send\_ok\_1$ and $1.receive\_ok\_2$

$1.send\_ok\_0$ and $0.receive\_ok\_1$,
we have

- $4.faulty = True \rightarrow \Box\neg < communication\_ok\_42 > T$
- $\Box\neg < communication\_ok\_42 > T \rightarrow \Box\neg < communication\_ok\_21 > T$
- $\Box\neg < communication\_ok\_21 > T \rightarrow \Box\neg < communication\_ok\_10 > T$
- $\Box\neg < communication\_ok\_10 > T \rightarrow \Box end\_ok = False$

So finally, we get: $4.faulty = True \rightarrow \Box end\_ok = False$
and the property of propagation is proved.

7 Conclusion

We proposed in this paper an hybrid formalism which improves the mixing of categories and logic proposed by Fiadeiro and Maibaum. We keep their categorical structure which has several advantages. At the specification level, the modularity induced by the categories allows to describe a big system more easily: each component is described independently (encapsulation); then the interactions between the components are specified; and finally, with composition laws a la CCS, we build a description of the system which is itself an object of the same category and can be reused. At the verification level, lemmas are verified locally on the components of the system (encapsulation) ; then these lemmas are translated along the morphisms to the global description. Thus, modular proofs can be achieved.

We add to this framework a logic well-suited for the design and verification of concurrent and reactive systems. Our logic inherits both from dynamic logic and temporal logic. So at the specification level, we have got on the one hand the notion of states which is very useful for the expression of properties and for the modeling of some static characteristics; and on the other hand, actions allow us to specify easily dynamic behaviors and to handle communications. Moreover, we gave a methodology to guide writing of specification in this logic. At the verification level, our logic is proved sound, complete and decidable and we hope this result makes easier the automation of proofs.

We applied this formalism to Modulor, a real-sized and fault-tolerant parallel machine. We thus realized our formalism was adapted to the specification and verification of such systems. As far as we know, there are few reports about the use of formal methods to validate fault tolerance mechanisms based on a passive redundancy approach. Most of the experiments deal with masking transient fault by active redundancy. In this context, we have only found logical proofs of correctness of vote or synchronization algorithms ([14]). The authors explain their logical approach is really expensive and can be applied fruitfully only to the most critical part of the system, to get generic results. Our modular way of specifying and validating can complement these approaches and extend their scope. Moreover, the logic proposed is well-suited for modeling passive redundancy: action and communication concepts are useful for modeling the behavior of a distributed system whereas the state notion caught in the logic makes easier the fault handling.

Acknowledgements We would like to thank P. Michel for his helpful comments.

References

1. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.
2. K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison-Wesley, 1988.
3. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *proceedings of the workshop on automatic verification methods for finite-state systems, LNCS 407*, pages 24 – 37, 1989.
4. V. David, Ch. Fraboul, J.Y. Rousselot, and P. Siron. Partitioning and mapping communication graphs on a modular reconfigurable parallel architecture. In *CONPAR'92*, Sept 1992.
5. E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science*, pages 996–1071. Elsevier Science, 1990.
6. J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 1992.
7. D. Harel. *Handbook of philosophical logic*, volume 2, chapter 10, Dynamic Logic, pages 497–604. 1984.
8. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
9. D. Kozen. Results on the propositional mu-calculus. *Theoritical Computer Science*, 27:333–354, 1983.
10. L. Lamport. The temporal logic of actions. Technical Report 79, SRC, 1992.
11. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.
12. Robin Milner. *Handbook of theoretical computer science*, chapter 19, Operational and algebraic semantics of concurrent processes, pages 1203–1242. Elsevier Science, 1990.
13. F. Pagani, C. Seguin, P. Siron, and V. Wiels. Verification experiments on a large fault-tolerant distributed system. In *Workshop AMAST "Model and Proof"*, Bordeaux, France, juin 1995.
14. John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 571 in LNCS, Nijmegen, The Netherlands, January 1992. Springer Verlag.
15. D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice Hall, 1988.
16. J. Sauloy. Interconnexion de modules. Technical report, CERT-ONERA, DERI, 1992.
17. V. Wiels. Specification et verification de programmes paralleles tolerants aux fautes. Master's thesis, E.N.S.E.E.I.H.T, 1994.