Modular approach to specify and validate an Electrical Flight Control System *
Category 1

M. Doche[1**], I.Vernier-Mounier[2], and F. Kordon[2]

[1] Department of Electronics and Computer Science
University of Southampton, Highfield
Southampton SO17 1BJ, United-Kingdom
mfd@ecs.soton.ac.uk

[2] Laboratoire d'Informatique de Paris 6, 4 place Jussieu,
F-75252 Paris Cedex 05, France
{ Isabelle.Vernier-Mounier, Fabrice.Kordon} @lip6.fr

Abstract. To study a part of an Electrical Flight Control System we have developed a tool-supported method dedicated to the incremental specification and validation of complex heterogeneous systems. Formal description of a system is structured in modules that interact.
We combine two modular approaches that share the same view of modularity but offer complementary validation procedures: model checking and functional test generation. We have adapted these validation procedures to take care of the modular aspects of our specification. They are performed incrementally. We first consider basic modules, then the communication between modules and finally composed modules.
To support our method, we have adapted existing tools, dedicated to non-modular specifications, to deal with modular constraints. These tools are integrated into a common platform to build a coherent execution environment.

Keywords. Heterogeneous Specification, Modularity, Verification, Test Generation, Case Tools.

1 Introduction

Critical Embedded systems must ensure fault tolerance requirements. They are more and more complex as their functions increase and become more sophisticated. These systems are structured in several heterogeneous components strongly interconnected. Components may represent software as well as hardware parts of the system. They may be specified in several specification languages. Components are often independently well-known and reused from one version of a

* Work supported by the *VaMoS* project, one of the four projects of the French action FORMA (http://www.imag.fr/FORMA/)

** This work was done while I was working at ONERA-CERT/DTIM, Toulouse.

system to a new one. The main difficulty lies in the number and the diversity of interactions between components. So, we present in this paper a tool-supported method to formally specify and analyze modular specifications of embedded systems. The integration of the tools in a common framework has allowed us to apply our method to a significant industrial application developed to study a part of an Electrical Flight Control System (EFCS). This industrial application is proposed by Sextant Avionique and is significant both for its size and complexity, representative of a wide range of embedded architectures. Specification, verification and test case generation results obtained on this system are given in the corresponding sections of the paper.

There exist well-tried methods and tools to specify and validate specifications. Their application to non-trivial software or process control remains difficult, but some recent results are promising [4, 27, 28, 10]. To overcome these difficulties modular specification and verification methods dealing with components are needed [12]. Problems appear with the specification of communications between components, decomposition of global properties into a set of properties that deal with a single component at a time, incremental specification and verification.

Several modular methods are proposed in the literature to verification [2, 3] or test generation [5, 26]. Our approach is to propose a set of modular methods to deal with each steps of the software development. We verify structural properties of the components to ensure that their compositions are possible and lead to the expected result. These constraints are expressed by the composition links, *morphisms*, between the two components. This allows us to separate the specification coherence verification from the behavioral properties verification and test case generation. Furthermore, if the specification coherence is verified, a module can also be reused several times.

We combine two modular approaches. MOKA [25] is a tool-supported method dedicated to the specification and composition of modules. The coherence between interfaces and different parts of modules is checked before composition. Combined with the metric temporal logic TRIO [21], we generate functional test cases at different level of abstraction [17]: basic components, interactions and cluster (i.e. the component that results of the composition of others). The OF-Class [14] language is dedicated to object oriented specification and behavioral verification. It allows the verification of each module independently, by computing an abstraction of the components that communicate with it, i.e. its *environment*. The two approaches offer complementary verification procedures and share a same view of the modularity. Therefore, it was valuable to integrate them in a unified modular method and to support it by integrating the two related sets of tools into the platform Framekit [23].

Section 2 describes our specification language, the *VaMoS* language and the *structural verification* of the coherence of our specifications.

Section 3 shows how to analyze the behavior of the model, by verifying constraints of the specified behavior.

In Section 4 we describe the generation of test cases at each level of abstraction we consider.

Moreover, we have integrated the different validation tools (structural verification, behavioral verification and test cases generation) into a platform presented in Section 5. A graphical interface helps to design modular specification and we ensure the coordination of the validation process with *note files*, which contain information on the state of the validation.

We conclude and present future works in Section 6.

2 The *VaMoS* language

VaMoS is a modular language to specify and to validate complex systems. A component is described in two steps. The first one deals with the modular aspects of the specification. We define the components of the system and their interactions with the environment. The second step is the description of the internal behavior of each component.

2.1 Modularity in *VaMoS language*

For complex systems with heterogeneous components, a modular specification highlights the interface of each component and its interaction with the environment. Then we can define links between components and build clusters of components according to modular operations.

Module A module is composed of four specifications. The parts *Param*, *Import* and *Export* constitute the interface of the module. The fourth part, *Body*, represents the internal behavior of the module.

- *Param* contains the parameters of a generic module.
- *Import* contains the declaration of the items of the environment needed by the module and the conditions under which these elements can be used.
- *Export* contains the declaration of the items supplied by the component and the conditions under which the environment can use them.
- *Body* is the internal part of the component. It specifies how the imported elements are used and how the exported ones are computed.

The four specifications are interconnected by four internal morphisms, which ensure safe formal connections between the specifications (cf. Figure 3 and 5). A morphism links an item of a *source* specification to an item of a *target* specification with two constraints: both the linked items have the same type and the behavior of the source specification is preserved in the target specification. The target specification is considered as a refinement of the source specification. In the particular case of a module, the specifications *Param*, *Import* and *Export* do not contain behavior description but properties which describe conditions on their items. So the second constraint means in this case that the properties of the interface can be deduced from the behavior specification of *Body*. The internal morphisms allow to verify the coherence of the four parts of the module and of its interface.

Reusability The explicit specification of *Import* and *Export* conditions allows us to:

- reuse a module. We just have to verify that the environment in which we want to plug the module respects the *Import* and *Export* conditions. This verification is performed independently of the implementation of the behavior. The verification of the coherence between the four parts of a module is performed only one time (cf. 2.3).
- refine the behavior of a module independently of its environment. Since the *Body* part is the only one to be modified, there is no modification of interfaces. Therefore, the coherence of the *Body* and the interfaces has to be verified (cf. 2.3).
- manage several implementation of a same module specification. The module instances differ only by the implementation of the *Body* part and the choice of the generic parameters.

Modular operations A cluster is the component that results from the composition of two others. The links between components and the interface of the resulting cluster depend upon relationships between these components.

Composition operations are defined by external morphisms that identify items of the interfaces of the composed modules. The four parts and the four internal morphisms of the cluster module are automatically computed. Thus, we incrementally build a *system module*, which entirely specifies the system. For more details on module concepts and calculus, see [25, 30].

Case study The presented case study is a part of an Electrical Flight Control System (EFCS). It is proposed by Sextant Avionique and is a significant industrial application both for its size and complexity, representative of a wide range of embedded architectures (for instance in A330-40 aircraft [7]). Hardware and software techniques must ensure fault tolerance requirements. Therefore the code correctness is highly critical ([16, 1]).

We consider here the part of the EFCS that manages the spoilers of an airplane during the takeoff and landing. The opening angles applied to the spoilers are computed with respect to the value given by sensors (altitude, speed, ...) and the angle the pilot wants to apply.

A function **Fangle** computes an angle that depends on two parameters. One parameter is the angle the pilot wants to apply to the spoilers, it is identified by the position of a lever and confirmed by a sensor. The other one is a signal that indicates if the airplane is on ground. This signal, *secured ground*, is secured by redundancy of the function dedicated to its computation. A decision procedure

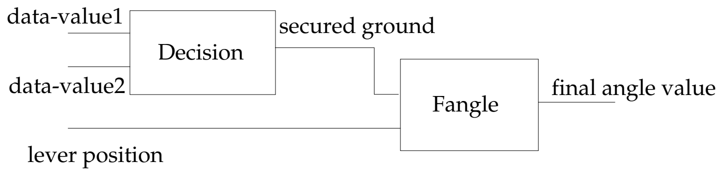

Fig. 1. Modular composition of the case study

compares the two results and transmits their common value if they match or a pre-defined one otherwise. Figure 1 describes this system.

We specify this mechanism by composition of two modules: **Decision** and **Fangle**. Then we define the interfaces of the modules:

- The **Decision** module imports two signal values (data_value1 and data_value2) computed from values of the sensors by two components that are not represented and computes and provides a value (secured_data) of the detection of the presence of the airplane on ground.
- The module **Fangle** imports a lever position value (lever_value) provided by the environment and the command of the pilot, a signal (secured_signal) provided by the module **Decision** and computes an angle value (final_angle) to apply to spoilers.

We specify now the relationships between the components. The module **Decision** and the module **Fangle** are composed by a *partial composition* operation. The module **Decision** is a supplier and the module **Fangle** is a user. The *Export* of the module **Decision** satisfies partially the *Import* of the module **Fangle**. The external morphism h identifies the secured_data exported by **Decision** with the secured_signal imported by **Fangle**. Figure 2 shows the four parts of each module.

The cluster module, represented by Figure 3, imports the lever position and the two values that are imported by **Decision**. It exports the angle value computed by **Fangle**. Its internal morphisms are automatically computed.

In such a case study, where redundant mechanisms appear often, the genericity and reusability capabilities of our formalism are very useful: for the whole case study, composed of 18 components, we have specified only 6 modules.

2.2 VaMoS Specification Language

We show how behavioral specifications are handled within modules. Each part of a *VaMoS* module contains :

- The declaration of the names of items used to describe the behavior of the system and the prototype of actions. It is denoted *Vocabulary*.
- The logical description of constraints on items of the interface for the *Param*, *Import* and *Export* parts. For the *Body* part it contains the logical description of the behavior. It is denoted *Formulae section*.

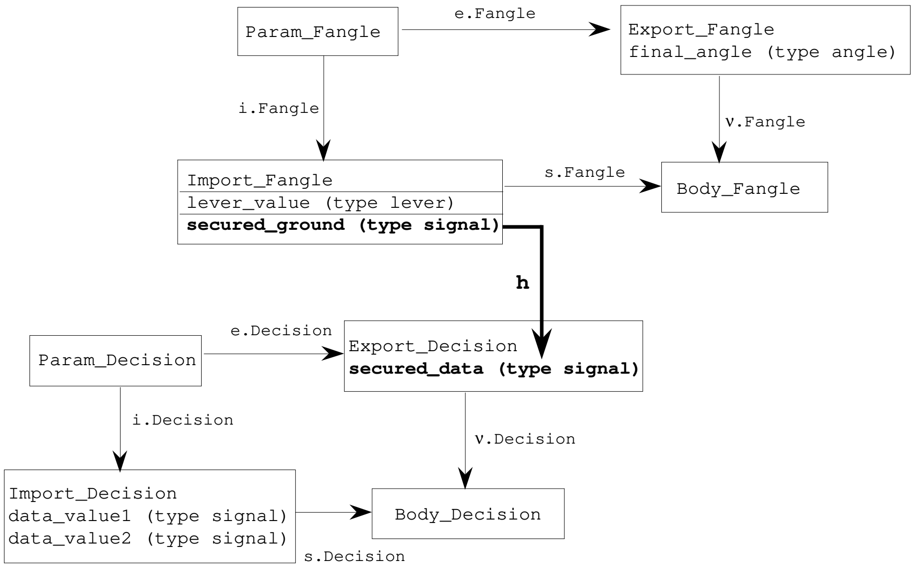

Fig. 2. Definition of the partial product of modules Fangle and Decision

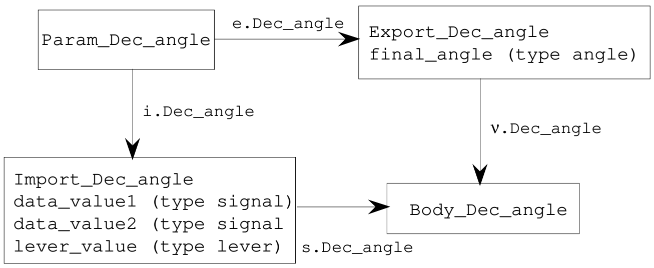

Fig. 3. Result of the partial product of modules Fangle and Decision

– The description of the actions performed by the module for the *Body* part only. It identifies the actions, explicitly describes them as well as their control system. This description is denoted *Imperative language section.*

The *Formulae* and *Imperative language* sections of the *Body* part allow us to manage conjointly logical and behavioral views of a same component.

The language used to specify the *Formulae section* is a linear temporal logic. We have chosen the TRIO logic. The TRIO logic provides a description well adapted to test case generation. A set of TRIO logic formulae represents the logical links between the actions and the properties they must satisfy.

The language used to specify the *Imperative language section* is a C-like language (automatically translated into Colored Petri Nets (CPN)). Properties are expressed apart from the behavior description in the TRIO logic. Therefore, they can be verified by model checking.

Vocabulary The vocabulary consists of four declaration parts:

– *Sorts* is a set of types;
– *Constants* declares constant functions;
– *Variables* declares variable functions, which describe the state of the system;
– *Actions* models occurrences of events operating on the variables of the system.

The example below shows the vocabulary part of the *Body* specification of the **Decision** module. The component operates on data of type $Decision\_signals$. The specific signal values are t (true) and f (false). The state of the component is described by three signals $Decision\_secured\_data$, $Decision\_data\_value1$, and $Decision\_data\_value2$. These component variables are modified when one of the actions $Decision\_imp\_data\_value1$, $Decision\_imp\_data\_value2$ or $Decision\_compute\_secured\_data$ occurs.

$Specification\ Decision =$
$\quad Sorts : Decision\_signals;$
$\quad Constants :$
$\quad\quad t : Decision\_signals;$
$\quad\quad f : Decision\_signals;$
$\quad Variables :$
$\quad\quad Decision\_secured\_data : Decision\_signals;$
$\quad\quad Decision\_data\_value1 : Decision\_signals;$
$\quad\quad Decision\_data\_value2 : Decision\_signals;$
$\quad Actions :$
$\quad\quad Decision\_imp\_data\_value1(Decision\_signals);$
$\quad\quad Decision\_imp\_data\_value2(Decision\_signals);$
$\quad\quad Decision\_exp\_secured\_data(Decision\_signals);$
$ImperativeLanguage : ...$
$Formulae : ...$
$endSpecification$

An occurrence of the action $Decision\_imp\_data\_value1$ (respectively $Decision\_imp\_data\_value2$) allows the acquisition of a unique value, which is stored in the local variable $Decision\_data\_value1$ (respectively $Decision\_data\_value2$). The local variable $Decision\_secured\_data$ takes the value t if and only if the both previous variables take the value t.

Imperative Language The OF-Class [13] language is dedicated to modular specification. Modules are composed regarding the interface specification of the modules. From the *Imperative language section* ,the OF-Class compiler automatically generates a Colored Petri net from one or several components [14].

The compiler represents each variable by a place. Interfaces places represent imported and exported variables. Actions are represented by transitions. The composition of modules is performed by identification of interfaces places.

Figure 4 represents the significant elements of the Petri net automatically generated from the specification of the **Decision** module. The declaration part of the CPN defines the classes (types) associated with the places (variables) used by the specification. The double-circled places represent the interface variables. The black transitions are the actions identified in the vocabulary declaration; they represent the importation and exportation actions. The two transitions *compute_t* and *compute_f* represent the two possible cases to compute the secured data. The OF-Class compiler adds information to represent the control of the variables. The complete automatically computed CPN holds 11 transitions, 16 places and 46 arcs.

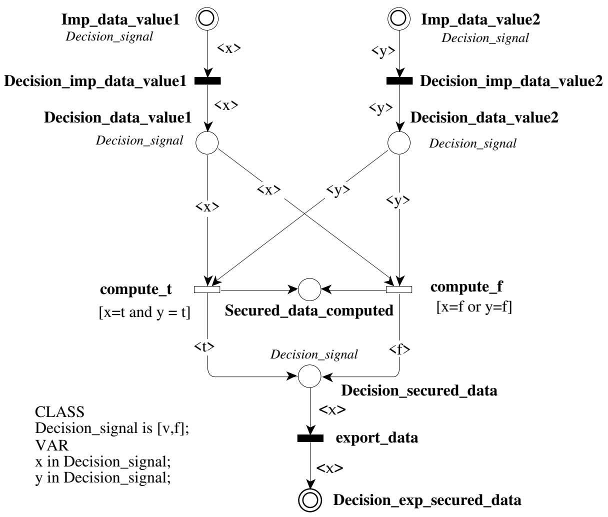

Fig. 4. Petri net

Formulae The part *Formulae* of a *VaMoS* specification contains TRIO formulae generated from vocabulary items, and logical connectors. TRIO ([21]) is a first order linear temporal logic. A TRIO formula is composed of atoms and logical connectors: classical ones (& and, | or, $\sim$ not, $\rightarrow$ implies, $\leftrightarrow$ equivalent), quantifiers ($\forall$ for all and $\exists$ exists) or temporal operators. Moreover, TRIO language holds the basic temporal operator : $Dist(F, \delta)$ means the formula F is true at δ time units from the current instant (δ can be positive or negative).

The TRIO operators allows us to derive all the classical linear temporal operators, in particular the following ones:

- *Next state(F)* is true iff F is true in the next state.
- *AlwF(F)* is true means F is true in all the following instants.
- *SomF(F)* is true if F means true in at least one of the following instants.
- *Until(F,G)* is true if F means true until G becomes true.

To describe the behavior of the system, we define a set of axioms, each of them is a TRIO formula. In the example we have chosen to associate an axiom to each condition on an action:

- $Decision\_signal\_availability\_value1$ (respectively $Decision\_signal\_availability\_value2$) states how the action $Decision\_imp\_data\_value1$ (respectively $Decision\_imp\_data\_value2$) occurs at each time instant.
- $Decision\_local\_copy\_value1$ (respectively $Decision\_local\_copy\_value2$) asserts that, each time a signal is acquired, its value is stored in the adequate local variable.
- $Decision\_secure\_result$ specifies how the secured local value $Decision\_secured\_data$ is computed using the value of the other local parameters.
- $Decision\_publish\_secured\_result$ characterizes the action $Decision\_exp\_secured\_data$.
- At each time instant, the action publishes one and only one secured value: $Decision\_secured\_data$.

The following set of formulae is the TRIO specification of these axioms.

$$
\begin{array}{l}
Specification\ Decision = ...\\
Formulae:\\
\quad Vars:\\
\quad\quad x, y : Decision\_signal;\\
\quad Axioms:\\
\quad\quad Decision\_signal\_availability\_value1:\\
\quad\quad\quad Alw(\exists\ x\ (Decision\_imp\_data\_value1(x) \&\\
\quad\quad\quad\quad \forall y(Decision\_imp\_data\_value1(y) \rightarrow x = y)));\\
\quad\quad Decision\_signal\_availability\_value2:\\
\quad\quad\quad Alw(\exists\ x\ (Decision\_imp\_data\_value2(x) \&\\
\quad\quad\quad\quad \forall y(Decision\_imp\_data\_value2(y) \rightarrow x = y)));\\
\quad\quad Decision\_local\_copy\_value1:\\
\quad\quad\quad Alw(\forall\ x\ (Decision\_imp\_data\_value1(x) \rightarrow Decision\_data\_value1 = x));\\
\quad\quad Decision\_local\_copy\_value2:\\
\quad\quad\quad Alw(\forall\ x\ (Decision\_imp\_data\_value2(x) \rightarrow Decision\_data\_value2 = x));\\
\quad\quad Decision\_secure\_result:\\
\quad\quad\quad Alw(Decision\_secured\_data = t \leftrightarrow\\
\quad\quad\quad\quad Decision\_data\_value1 = t\ \& Decision\_data\_value2 = t);\\
\quad\quad Decision\_publish\_secure\_result:\\
\quad\quad\quad Alw(\forall\ x\ (Decision\_exp\_secured\_data(x) \leftrightarrow\\
\quad\quad\quad\quad x = Decision\_secured\_data));\\
endSpecification
\end{array}
$$

2.3 Structural verification

A preliminary step of validation (we call it "structural verification") verifies the consistency of a modular specification based on the modules and the composition operations presented in section 2.1.

For each module, we check the four internal morphisms, which means for each morphism :

- The source vocabulary is included in the target vocabulary depending on morphism and two linked items have the same type. The MOKA tool, developed at ONERA-CERT ([25, 30]), performs type checking of the morphisms.
- The constraints defined in the specifications of the interface can be deduced from the *Body* specification :
  - In CPN, the properties that represent the constraints are verified on the Petri net specification by model checking.
  - In TRIO, we prove that axioms of the source can be deduced from the set of axioms of the target. The MOKA tool only generates a "*proof obligation*" that has to be proved by some other tool (for example a prover on TRIO logic).

Then, for each specification we check the consistency of both description parts by classical means (parsers and type checkers dedicated to each language).

For modular operation definition, to control the validity of each interconnection, we must check the external morphisms as we check internal morphisms.

This verification, performed by the MOKA tool, leads to the automatic computation of the four parts of the cluster module.

This ensures that what is needed by a module is offers by the one with which it is composed. Several verification, as behavioral or test case generation, are needed to ensure that what is offered is exactly what was expected.

3 Behavior verification of modular specifications

To verify properties we use a model checker. The first step is to decompose the property into a set of local properties and a set of communication ones. Each local property concerns the internal behavior of a module. Its verification can be performed independently of the other modules. The communication properties express relations between modules. Before their verification, the composition of the concerned components must be computed.

3.1 Modular aspects

The CPN, generated by the OF-Class compiler, are dedicated to modular verification. Each module is viewed as a function that imports data, computes results and exports them. The computation of results is an internal action. Therefore, once a module has collected all the needed values, its environment has no impact on the way the results are computed. That is why all the properties that express properties on the computation step are preserved by composition. Such are properties that express links between the values of the imported data and the value of the results.

To independently study each module, a representation of its environment is computed when it is translated into a Petri net. It represents the production of all the possible imported data and the use of the exported ones. There is no restriction on the possible values of the imported data and they are provided each time they are needed. Therefore, no deadlock results from this specification of the environment and it does not restrict the possible behaviors of the module. This representation is used to verify local properties that deal with the relation between the values of the imported data and the exported ones or internal deadlocks. If a local property is verified in this representation, it is verified whatever the environment of the module may be. But, if the property is not verified, no conclusion is possible. The represented environment may produce *bad* values that are not produced by the real environment. To obtain a certitude, we consider the module in its real environment.

To verify properties that concern the communications between several modules, we have to compose them. Such a property may be the detection of deadlocks of the system. In this case the composition of the whole system is necessary. The environment of several modules is represented by the same way as for one module. Therefore, incremental verification is possible. In [4] the authors model the abstraction of the environment to check a part of a case study, but this is manually performed.

3.2 Abstractions

Despite important results in the representation of the state space [9], the state space explosion problem happens even for one module. Therefore we have worked on abstractions of the specification.

The first abstraction we have identified is due to the specification of data with a large set of possible values (speed, altitude, ...). Very often, given a variable with a large domain, each possible value of the variable does not lead to a different set of instructions to execute. The domain is decomposed into sub-domains that lead to different sets of instructions.

The second abstraction depends upon the property to verify. Instructions and properties without incidence on the result of the verification are suppressed. Rules to perform abstractions and simplifications have been identified and their implementation is in progress. These rules have been applied on the example above leading to a reduction factor of 100 to 1000 depending upon the verified properties. The same principle is applied in [10]

The computation of abstractions is not yet tool supported but rules have been identified and applied on the example presented in the paper. The state space reduction we obtain is significant.

3.3 Practical application

CPN [22] are well adapted to describe the control of systems and to support behavior verification. We use the model checker PROD[1]. It is a Petri net reachability analysis tool that supports on the fly verification of linear time temporal properties as well as verification of branching time properties. Moreover, linear temporal logic formulae supported by PROD can be expressed in TRIO logic. This allows us to complete or to confirm this verification step with tools dedicated to TRIO language.

3.4 Example

The redundancy introduced to support failures needs some adaptation of the software to manage the fact that two identical functions do not give the same result. In our example, module **Fangle** must provide a neutral angle value (i.e., 0 value) if module **Decision** imports two different values. The temporal specification of this property is:"at each instant, (module **Decision** imports two different values) implies (there is an instant in the future such that module **Fangle** exports 0)". It is a liveness property. The TRIO formulae are:

$$\begin{aligned}
&CommunicationProperty \equiv AlwF(DecisionProp \Rightarrow SomF(FangleProp))\\
&DecisionProp \equiv \exists x1, x2\ Decision_imp_data_value1(x1)\ \&\\
&\qquad\qquad\qquad\qquad Decision_imp_data_value2(x2)\ \&\ (x1 \neq x2)\\
&FangleProp \equiv \forall sp\ Fangle_compute_angle(sp, 0)
\end{aligned}$$

[1] PROD is developed at the Helsinki University of Technology ([29])

We decompose $CommunicationProperty$ in three lemmas that highlight the signal exported by module **Decision** and imported by module **Fangle**. Lemma L1 concerns module **Decision**. It shows the consequence of *DecisionProp* on the value of the exported signal. Lemma L2 concerns module **Fangle**. It shows the conditions that must be satisfied by the imported signal to ensure *FangleProp*. Lemma L3 concerns the relationship between the modules. It ensures that if module **Decision** exports the signal value f, module **Fangle** imports the same value.

$CommunicationProperty \equiv (L1 \ \& \ L2 \ \& \ L3)$
$L1 \equiv AlwF(DecisionProp \Rightarrow Decision\_exp\_secured\_data(\mathrm{f}))$
$L2 \equiv AlwF(Fangle\_imp\_data\_signal(\mathrm{f}) \Rightarrow SomF(FangleProp))$
$L3 \equiv AlwF(Decision\_exp\_secured\_data(\mathrm{f}) \Rightarrow Fangle\_imp\_data\_signal(\mathrm{f}))$

Initial specification The state space of the module **Decision**, computed in 16 seconds, holds 80 states and 93 arcs. Property L1 has been verified in 26 seconds on it. The state space of module **Fangle**, computed in 4 minutes and 8 seconds, holds 9,247 states and 19,250 arcs. The state space of the global system, computed in 18 hours 54 minutes 20 seconds, holds 818,390 states and 2,605,318 arcs

Abstraction of data domains The module **Fangle** uses angle variables that belong to the interval [0..46]. The variables of this domain are only compared with the maximum value. As the values belonging to [0..45] lead to the execution of the same instructions we have mapped the interval on [0..1]. The values [0..45] are identified with 0 and 46 with 1. This does not modify the possible behaviors of the components. Of course, such an abstraction may not be applied for a property dealing with the exact value of a variable in the domain. The reduced state space of module **Fangle**, computed in 16 seconds, holds 113 states and 135 arcs. Property L2 has been verified in 27 seconds on it.

Abstraction regarding property L3 This abstraction is significant for the property that ensures that if *Decision* exports the value "f" then **Fangle** imports the same value. The way the values are computed is not important, only the communication instructions affect the property. We reduce the size of the specification by deleting the instructions that do not affect the communications between the two modules. This abstraction is applied jointly with the one on the data domains. The state space of the global system has been computed in 27 seconds. It holds 176 states and 318 arcs. Property L3 has been verified in 1 minute and 11 seconds.

4 Test case generation from modular specifications

To complete the validation process, we also generate functional test cases.

Many works deal with test cases generation from non-modular formal specifications [20, 8], but many techniques are limited by the size of the specifications. We take advantage of the modular structure of our specifications to assist the test cases generation at different levels of abstraction:

- First, *unit level* tests independently and in details little parts of the system or basic components. The generation is based on basic modules that describe the basic components.
- Then, *integration level* tests the interactions between components of the system. For this step, we focus on interfaces of modules and morphisms defined in a modular operation to describe the links between the components.
- Finally, *cluster level* allows to detect global errors of the system. The generation is achieved by composition of test cases from basic modules according to the modular operations.

Moreover, we will generate, at each step, a correct test set as defined in [6]. A set of test cases is *unbiased* if it does not reject a correct program and it is *valid* if it accepts only correct programs.

4.1 Modular aspects

In our approach, we want to reuse as often as possible classical generation techniques from non-modular specifications, which are often based on the generation of models of the specification (see [20, 8] for a description of different techniques). But during the generation process from modular specifications, new constraints appear due to the modular aspects:

Encapsulation of data : at unit and cluster level, we want to generate test cases that contain only data of the interface.

Renaming according to the morphisms : at integration level, we want to base the generation on the interface of several modules and on morphisms which describe links between the items of the interfaces.

Composition according modular operations : at cluster level, we want to reuse test cases defined at unit level and to compose them to obtain test cases for the cluster.

In the sequel, we describe briefly the processes follow to deal with encapsulation, renaming and composition; we illustrate them on our example (more details can be found in [17, 15]). How these processes preserve correctness is described in [18, 15].

Encapsulation A module encapsulation allows to hide non-visible items. These items must neither be observed nor commanded during functional test steps. However the test cases generated from the module are possible executions of its internal specified behavior. To deal with test generation at unit level:

- We generate test cases from the *Body* specification of the module. Indeed, by construction and structural verification, this part contains the complete description of the behavior of the module and constraints on its interface. For this purpose, we use existing test case generation method from non-modular formal specification.
- We project the resulting test cases on the vocabulary of the interface (parts Export, Import and Param) such that the new test cases contain only visible items.

Assuming that the structural verification has succeeded, if we succeed to generate a correct test set from the Body specification, its correctness is preserved during the projection step. Indeed, projection step reduces items of the test cases, so they accept at least all programs accepted before projection step and an unbiased test set remained unbiased. Its validity is preserved due to conditions on the internal morphisms and structure of the module (see [15]).

Renaming Interactions between components are described by a set of morphisms between the interfaces of the modules, according to a modular operation. To verify the modular specification, we generate and prove proof obligations on these morphisms (see section 2.3), which allow to check that the behavior of the target specification of a morphism maps the behavior of its source specification. We need to make the same check on the implementation: each possible behavior of the target specification meets the constraints stated by the source specification. So to deal with integration level, we follow for each morphism of a modular operation this process:

- We generate test cases from the source specification of the morphism by a classical non-modular technique. They describe the constraints stated by this specification.
- We rename each test case according to the morphism, to obtain test cases defined on the vocabulary of the target, for the constraints of the source specification.

Validite and unbiased features of the build test set are preserved during renaming step because of the properties of the morphism. For more details of these approaches, see [18].

Composition Application of a module composition creates a new module (see section 2.1). A naive but inefficient approach in practice would be to incrementally generate the global system and to generate test cases for it. Therefore, we reuse the test cases generated for unit level and we rely on properties of modular operations to complete the test cases.

For example, let us consider the partial composition described in section 2.1. The TRIO tool generates two sets of test cases for the user module $Fangle$ and for the supplier module *Decision*. We merge, by pair, any test case of $Fangle$

with any compatible test case of *Decision* to obtain a test case of the resulting module. Compatibility means that all common items of the both modules have the same evaluation at a given instant in the both test cases. That will be illustrated in the following example.

Once again, correctness is ensured by the correct definition of the cluster module and the external morphisms between both basic modules. A priori generated test cases accept at the most programs accepted by the test sets of both basic modules, so validity is preserved. Internal structure of the basic modules and external morphisms between these modules ensure that the generated test set remains unbiased. For more details of these approaches, see [18].

To achieve test case generation at cluster level, this composition step must be followed by a projection step to obtain test cases defined on the interface of the cluster.

4.2 Practical application

To generate correct test cases from the specifications of a module, we use the TRIO Model generator[2]. It is founded on the TRIO language, and it allows generation of temporal finite partial models (called *histories*) from a specification. We define a *temporal window*, which is an interval of integer, and which represents the time scale [21]. Each history is considered as an abstract test case and is composed with a set of pertinent evaluation of items at any instant of the temporal window.

To achieve an entirely tool supported method, we develop tools to deal with modular aspects: encapsulation, renaming and composition [15].

4.3 Example

Test cases generation from the body specification Let us consider the module **Decision**: for a temporal window of two units the TRIO model generator has generated 16 test cases in 3.9 seconds.

Encapsulation of data on an unitary test case We project each test case generated from the body on the vocabulary of the interface. The projection of the test case c generated from the Body of **Decision** on the vocabulary of the interface of **Decision** leads to the test case k:

[2] A TRIO formal specification environment for complex real-time system specifications has been developed by CISE, supported by Politechnico Di Milano, under a contract of ENEL/CRA ([11])

$$c = \begin{pmatrix} Decision\_secured\_data = f & : 1 \\ \sim Decision\_exp\_secured\_data(t) & : 1 \\ Decision\_exp\_secured\_data(f) & : 1 \\ Decision\_data\_value1 = f & : 1 \\ \sim Decision\_imp\_data\_value1(t) & : 1 \\ Decision\_imp\_data\_value1(f) & : 1 \\ Decision\_data\_value2 = t & : 1 \\ \sim Decision\_imp\_data\_value2(f) & : 1 \\ Decision\_imp\_data\_value2(t) & : 1 \end{pmatrix} \rightarrow k = \begin{pmatrix} \sim Decision\_exp\_secured\_data(t) & : 1 \\ Decision\_exp\_secured\_data(f) & : 1 \\ \sim Decision\_imp\_data\_value1(t) & : 1 \\ Decision\_imp\_data\_value1(f) & : 1 \\ \sim Decision\_imp\_data\_value2(f) & : 1 \\ Decision\_imp\_data\_value2(t) & : 1 \end{pmatrix}$$

"$Decision\_secured\_data = f \ : \ 1$" means that the variable $Decision\_secu$-red_data takes the value f at the instant 1.

Generation of integration test cases Assume s is a test case of the source specification of the morphism h (a part of the Import of **Fangle**). After renaming according to h, we obtain the test case t, defined on the vocabulary of the Export of **Decision**:

$$s = \begin{pmatrix} \sim Fangle\_imp\_secured\_ground(t) & : 1 \\ Fangle\_imp\_secured\_ground(f) & : 1 \\ \ldots & \end{pmatrix} \rightarrow t = \begin{pmatrix} \sim Fangle\_imp\_secured\_data(t) & : 1 \\ Fangle\_imp\_secured\_data(f) & : 1 \\ \ldots & \end{pmatrix}$$

Generation of cluster test cases Assume l is a projected test case of **Fangle**; it merges the test case k to obtain m. Actions that are linked by an external morphism are identified in m. It is possible only if they have the same value. Otherwise, the test cases cannot be merged. Actions of l and k that are not linked by the external morphism appear in m. In our example, the external morphism h identifies the action $Decision\_exp\_secured\_data$ of the Export of **Decision** to the action $Fangle\_imp\_secured\_data$ of the Import of **Fangle**:

$$\begin{matrix} k \\ + \\ l = \begin{pmatrix} \sim Fangle\_imp\_secured\_data(t) & : 1 \\ Fangle\_imp\_secured\_data(f) & : 1 \\ Fangle\_imp\_lever\_data(f) & : 1 \\ \ldots & \end{pmatrix} \end{matrix} \rightarrow m = \begin{pmatrix} \sim Decision\_exp\_secured\_data(t) & : 1 \\ Decision\_exp\_secured\_data(f) & : 1 \\ \sim Decision\_imp\_data\_value1(t) & : 1 \\ Decision\_imp\_data\_value1(f) & : 1 \\ \sim Decision\_imp\_data\_value2(f) & : 1 \\ Decision\_imp\_data\_value2(t) & : 1 \\ Fangle\_imp\_lever\_data(f) & : 1 \\ \ldots & \end{pmatrix}$$

Conversely, the following projected test case k' of **Decision** cannot be merged with the test case l because l and k' disagree on the evaluation of action $Deci$-$sion\_exp\_secured\_data$ and $Fangle\_imp\_secured\_data$.

$$k' = \begin{pmatrix} \sim Decision\_exp\_secured\_data(f) & : 1 \\ Decision\_exp\_secured\_data(t) & : 1 \\ \sim Decision\_imp\_data\_value1(t) & : 1 \\ Decision\_imp\_data\_value1(f) & : 1 \\ \sim Decision\_imp\_data\_value2(f) & : 1 \\ Decision\_imp\_data\_value2(t) & : 1 \end{pmatrix}$$

5 Implementation of a CASE environment

To efficiently test our methodology and build a coherent execution environment we have integrated the required tools into FrameKit [23]. Framekit is a generic platform dedicated to the rapid prototyping of CASE environment. Its implementation follows the guidelines of the ECMA-NIST reference model [19]. In FrameKit, presentation and display of services are strongly constrained. A polymorphic editor engine, Macao [24], provides a unified look and feel for the manipulation of models as well as access to services integrated in FrameKit. FrameKit manages three kinds of entities: *formalisms*, *models* and *services*. A formalism describes representation rules of a knowledge domain. A model is the description of a given knowledge using a formalism; it is a "document" composed with objects defined in the formalism. A service is a tool function that corresponds to operations in a design methodology. For example *VaMoS* modules and CPN are formalisms for which the user may define several models. The MOKA verifier, the TRIO test case generator and the Petri net model checker PROD are services. FrameKit allows the use of several formalisms. Furthermore, it manages shared data, versions of model specifications and provides good facilities for fast integration of new tools that were not initially designed for it.

5.1 Multi-formalism management

Parameterization of Macao and the services management of Framekit allows us to specify and handle multiple formalisms.

Editor and User Interface Macao is parameterized using external files that describe components of the formalism. Thus, the construction of a new formalism does not imply any recompilation. Of course, Macao deals with syntactical aspects only, semantical ones are a convention between the user and the tool.

Figure 5 shows the *VaMoS* formalism that is the graphical representation of the specification language presented in section 2.1. The figure shows the four nodes that represent the four parts of a module, the four internal morphisms that link them and the labels associated with the body part.

Services management Each service is relevant for an identified set of formalisms. FrameKit holds an instantiation mechanism that identifies the formalism of a model and creates the list of services dedicated to it. Therefore, the user can only ask for services that are relevant for its specification.

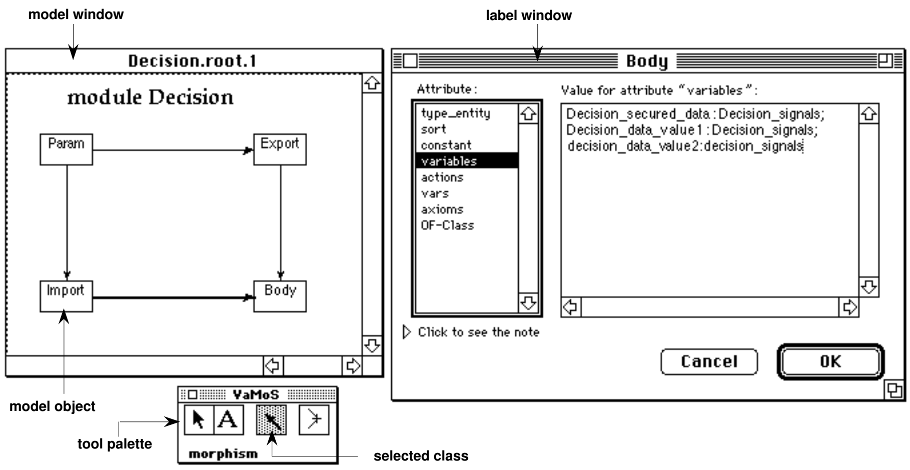

Fig. 5. VaMos formalism

5.2 Open Platform

FrameKit is an open platform that may be enriched by new services. To achieve this enrichment, a procedure called integration has been defined. We distinguish two types of tool integration : *a priori* and *a posteriori*.

The *a priori* integration concerns tools that are especially designed to run in the FrameKit environment. The compiler from OF-Class to CPN was developed to be integrated in Framekit; it was implemented to an *a priori* implementation.

The *a posteriori* integration concerns already designed tools (sometimes, source files may not be available) to be integrated in the FrameKit environment. It requires an adaptation of the imported software. A translation of the FrameKit file format into the file format expected by the tool is necessary. The opposite translation is necessary to store results. Moreover, for interactive tools, as MOKA and PROD, functions that drive the user interface are provided. MOKA, PROD and TRIO are *a posteriori* integrated tools.

Figure 6 shows how the integrated tools are linked. If some syntax errors or interface incoherence are detected, the specification must be modified. No verification tool can be applied.

5.3 Note files

A note file is attached to each specification module. It is a structured file containing all the information related to the analysis of the model. A note is associated to a property and gives:

– its identification,
– the method used to verify it,
– the part of the module concerned by the verification,

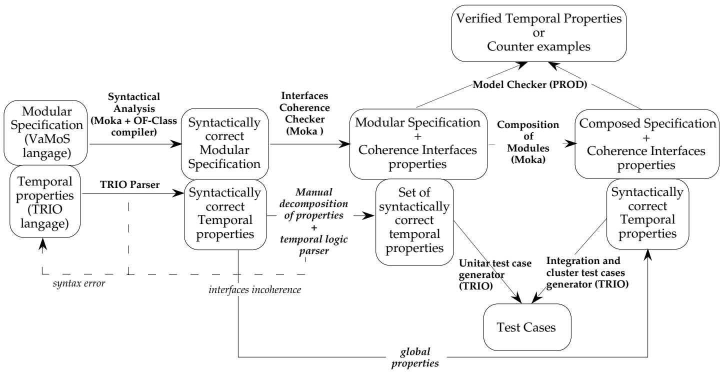

Fig. 6. Modular specification and validation methodology

- the way the property has been identified (by the model checker, the interfaces coherence checker, ...),
- the status of the verification (done, in progress, to do),
- the context of the verification (abstraction of the model, computation of the state space, ...).

The defined syntax of some note attributes allows the exploitation of the same note file by several tools. These *note files* may be used to produce an analysis report at each moment. The files associated with a model may be shared between several tools.

6 Conclusion

We have presented a tool-supported approach dedicated to the specification and validation of critical embedded systems. We deal with the complexity of the system using a modular methodology, which provides generic and reusable modules. Our homogeneous specification language allows us to manage conjointly two views of a same specification: a logical one and a behavioral one. Each of them is well adapted to a specific verification procedure (model checking and test cases generation). We are now identifying the common semantic aspects of Petri nets and Moka components to strengthen the links between both views.

The modular structure of our specifications is exploited to perform the verification of the system. We have improved these approaches to deal with a realistic industrial application ([1]). We are implementing a CASE environment integrating these tools into the platform Framekit. This allows us to trace the development process in both views, with note files to exchange information and version management. However, some steps of the validation methods are not yet tool supported. To improve these points, the semantics aspects can help us to increase the cooperation between the two approaches. Especially, we will study how validation tools can interact and how to use results of one step in the other. For example, TRIO model generator automatically computes data domain abstractions that are relevant for the model checker PROD. The model checker PROD defines possible execution scenarios that can be used to define sequences of test cases to test on the system.

References

1. Action FORMA. *Maîtrise de systèmes complexes réactifs et sûrs*, Journée au MENRT: Bilan de la 1^{ere} année, Paris, January 1998. http://www.imag.fr/FORMA/.
2. R; Alur, T.A. henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha : Modularity in model checking. In *proceedings on the 10th International Conference on Computer-Aided Verification*, pages 521–525. Springer Verlag, 1998.
3. H.R. Andersen, J. Staunstrup, and N. Maretti. A comparison of modular verification techniques. In *Proceedings of FASE'97*. Springer Verlag, 1997.
4. R.J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. In *Proceedings of the 4th ACM SIGOFT Symposium on the Foundations of Software engineering*, pages 156–166, 1996.
5. S. Barbey, D. Buchs, M-C. Gaudel, B. Marre, C. Péraire, P. Thévenod-Fosse, and H. Waeselynck. From requirements to tests via object-oriented design. Technical Report 20072, DeVa ESPRIT Long Term Research Project, 1998. http://www.laas.research.ec.org/deva/papers/4c.pdf.
6. G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6, November 1991.
7. D. Brière and P. Traverse. Airbus a320/a330/a340 electric flight controls: a family of fault-tolerant systems. *FTCS*, 23:616–623, 1993.
8. E. Brinksma. Formal methods for conformance testing: Theory can be practical. In *CAV'99*, number 1633 in LNCS, pages 44–46. Springer Verlag, July 1999.
9. J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. DILL. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 13*, 4:401–424, 1994.
10. W. Chan, R.J. Anderson, P. Beame, and D. Notkin. Improving Afficiency of Symbolic Model Checking for state-Based System Requirements. In *proceedings of the 1998 International Symposium on Software Testing and Analysis* , 1998.
11. E. Ciapessoni, E. Corsetti, M. Migliorati, and E. Ratto. Specifying industrial real-time systems in a logical framework. In *ICLP 94 - Post Conference Workshop on Logic Programming in Software Engineering*, 1994.
12. E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. Technical report, Carnegie Mellon University, 1996.
13. A. Diagne. *Une Approche Multi-Formalismes de Spécification de Systèmes Répartis: Transformations de Composants Modulaires en Réseaux de Petri*. Thèse, LIP6, Université Paris 6, 4, Place Jussieu, 75252 Paris Cedex 05, May 1997.
14. A. Diagne and F. Kordon. A multi-formalisms prototyping approach from conceptual description to implementation of distributed systems. In *Proceedings of the 7th IEEE International Workshop on Rapid System Prototyping (RSP'96)*, Porto Caras, Thessaloniki Greece, June 1996.

15. M. Doche. *Techniques formelles pour l'évaluation de systèmes complexes. Test et modularité.* PhD thesis, ENSAE, ONERA-CERT/DTIM, Décembre 1999.
16. M. Doche, J. Cazin, D. Le Berre, P. Michel, C. Seguin, and V. Wiels. Module templates for the specification of fault-tolerant systems. In *DASIA'98*, May 1998.
17. M. Doche, C. Seguin, and V. Wiels. A modular approach to specify and test an electrical flight control system. In *FMICS-4, Fourth International Workshop on Formal Methods for Industrial Critical Systems*, July 1999.
18. M. Doche and V. Wiels. Extended institutions for testing. In *AMAST00, Algebraic Methodology And Software Technology*, LNCS, Iowa City, May 2000. Springer Verlag.
19. ECMA. A Reference Model for Frameworks of Software Engineerings Environments. Technical Report TR/55 (version 3), NIST Report, 1993.
20. M-C. Gaudel. Testing can be formal, too. In *TAPSOFT'95*, pages 82–96. Springer Verlag, 1995.
21. C. Ghezzi, D. Mandrioli, and A. Morzenti. A model parametric real-time logic. *ACM Transactions on programming languages and systems*, 14(4):521–573, October 1992.
22. K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Volumes 1, 2 and 3.* Springer-Verlag, 1992.
23. MARS-Team. FrameKit Home page. http://www-src.lip6.fr/framekit.
24. MARS-Team. Macao Home page. http://www-src.lip6.fr/macao.
25. P. Michel and V. Wiels. A Framework for Modular Formal Specification and Verification. In *LNCS 1313, Proceedings of FME'97*, September 1997.
26. A. Morzenti, P. San Pietro, and S. Morasca. A tool for automated system analysis based on modular specifications. In *ASE98*, pages 2–11. IEEE Computer Society, 1998.
27. R. Pugliese and E. Tronci. Automatic verification of a hydroelectric power plant. In *LNCS 1051, FME'96: Industrial Benefit and Advances in Formal Methods, 3rd International Symposium of Formal Methods Europe*, pages 425–444, 1996.
28. T. Sreemani and J.M. Atlee. Feasibility of model checking software requirements: A case study. In *COMPASS'96, Proceedings of the 11th Annual Conference on Computer Assurance*, pages 77–88, 1996.
29. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. Technical Report ISBN 951-22-2707-X, University of technology, Departement of Computer Science, Digital Systems Laboratory, 1995.
30. V. Wiels. *Modularité pour la conception et la validation formelles de systèmes.* PhD thesis, ENSAE - ONERA/CERT, October 1997.