# Management of evolving specifications using category theory *

Virginie Wiels and Steve Easterbrook
NASA/WVU Software Research Lab
100 University Drive, Fairmont WV 26554, USA
{wiels,steve}@atlantis.ivv.nasa.gov

**Abstract**

Structure is important in large specifications for understanding, testing and managing change. Category theory has been explored as framework for providing this structure, and has been successfully used to compose specifications. This work has typically adopted a "correct by construction" approach: components are specified, proved correct and then composed together in such a way to preserve their properties. However, in a large project, it is desirable to be able to mix specification and composition steps such that at any particular moment in the process, we may have established only some of the properties of the components, and some of the composition relations. In this paper we propose adaptations to the categorical framework in order to manage evolving specifications. We demonstrate the utility of the framework on the analysis of a part of a software change request for the Space Shuttle.

## 1 Introduction

Structure is important in large specifications for the same reasons that it is important in programs: the principles of modularity and information hiding are essential for managing large-scale specifications [3]. A well-chosen structure greatly facilitates understanding, validation, and modification of a specification. In this paper, we are primarily concerned with the management of evolving specifications, and especially the analysis of change requests. Ideally, we would like a specification structure that allows us to isolate changes within a small number of components of a specification, and to reason about the impacts of a change on interconnected components.

Category theory has been proposed as framework for providing this structure, and has been successfully used to provide composition primitives in both algebraic [16, 3] and temporal logic [4] specification languages. Category theory is ideal for this purpose, as it provides a rich body of theory for reasoning

[*] This work is supported by NASA grant NAG 2-1134

about objects and relations between them (in this case, specifications and their interconnections). Also, it is sufficiently abstract that it can be applied to a wide range of different specification languages. Finally, it lends itself well to automation, so that, for example, the composition of two specifications can be derived automatically, provided that the category of specifications obeys certain properties (e.g. co-completeness). The drawback to category theory is that as an abstract branch of mathematics, it is even further removed from practical software engineering than most formal methods. Our philosophy has been to hide as much as possible of the underlying theory from the user, whilst providing an environment for interconnecting specifications, and reasoning about the resulting structures.

Work on category theory for software specification has typically adopted a "correct by construction" approach: components are specified, proved correct and then composed together in such a way to preserve their properties. However, in a large project, it is desirable to be able to mix specification and composition steps such that at any particular moment in the process, we may have established only some of the properties of the components, and some of the composition relations. This reflects the reality of large-scale specifications constructed by a team of people. Such specifications are inconsistent for most of their lifecycle. As the specification evolves, each change may introduce many inconsistencies. A "correct by construction" approach requires that these be eliminated before the change can be applied to the specification. In practice, it is often desirable to temporarily ignore the inconsistencies, for two reasons. Firstly, if resolution of an inconsistency depends on information that is not yet available, we wish neither to hold up the development process, nor to force a premature decision. Secondly, it is useful to be able to explicitly represent the inconsistencies, so that we can reason about possible corrective actions.

In this paper we demonstrate how the categorical framework can be adapted to manage evolving specifications. There are three elements to our approach: (1) the ability to deal with morphisms that are not completely correct (some proof obligations are not discharged); (2) the use of limits to provide information about the potential effects of different relationships on the system being built; and (3) the integration of properties in the same framework as the specifications, in order to facilitate their management and evolution.

The paper is structured as follows. Section 2 provides a brief overview of some notions of category theory, sufficient to understand the paper. Section 3 illustrates how the theory can be applied to the problem of composing specifications, using the category of specifications and specification morphisms. Section 4 describes our adaptation of this framework to deal with the problems of evolving specifications. Section 5 provides an example of the framework applied to the analysis of a part of a software change request for the Space Shuttle. Section 6 discusses implementations of the framework. Section 7 suggests avenues for further research.

## 2 Category theory

In this section, we give the definitions of some notions of category theory that we use in the remainder of the paper.

**Definition: category.** A category is composed of two collections:

- the *objects* of the category,
- the *morphisms* (arrows) of the category.

These two collections must respect the following properties:

- each morphism f is associated with an object A that is its domain and an object B that is its codomain. Notation: $f : A \to B$
- for all morphisms $f : A \to B$ and $g : B \to C$, there exists a composed morphism $g \circ f : A \to C$ and the composition law is associative, i.e. for all $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$
- for each object A of the category, there exists an identity morphism $id_A$ such that:
$$\forall f : B \to A, id_A \circ f = f$$
$$\forall f : A \to B, f \circ id_A = f$$

Category theory provides a framework to describe links between objects, and to manipulate them by means of operations. Here we describe two such operations: pushout and pullback.

**Definition: pushout.** A *pushout* of a pair of morphisms with same source $f : A \to B$ and $g : A \to C$ in a category is an object $D$ and a pair of morphisms $p : B \to D$ and $q : C \to D$ such that the square commutes (cf figure 1):

$$p \circ f = q \circ g$$

and such that the following universal condition holds: for all objects $D'$ and all morphisms $p' : B \to D'$ and $q' : C \to D'$ such that $p' \circ f = q' \circ g$, there exists a unique morphism $u : D \to D'$ such that $u \circ q = q'$ and $u \circ p = p'$.

Intuitively, the second part of the definition ensures that the $D$ chosen to construct the pushout is the "minimal" such $D$ amongst all the candidates $D'$.

The generalisation of this operation to several objects and morphisms is called a colimit. A practical interpretation for the colimit is given by Goguen in [5]:

"Given a species of structure, say widgets, then the result of interconnecting a system of widgets to form a super-widget corresponds to taking the colimit of the diagram of widgets in which the morphisms show how they are interconnected."

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "p"] \arrow[ddr, "p'"] & \\
C \arrow[r, "q"'] \arrow[drr, "q'"'] & D \arrow[dr, "u"] & \\
& & D'
\end{tikzcd}$$

Figure 1: Pushout of two morphisms f and g

**Definition: pullback.** A *pullback* of a pair of morphisms with same target $f : B \to A$ and $g : C \to A$ in a category is an object $D$ and a pair of morphisms $p : D \to B$ and $q : D \to C$ such that the square commutes (cf figure 2):

$$f \circ p = g \circ q$$

and such that the following universal condition holds: for all objects $D'$ and all morphisms $p' : D' \to B$ and $q' : D' \to C$ such that $p' \circ f = q' \circ g$, there exists a unique morphism $w : D' \to D$ such that $q \circ w = q'$ and $p \circ w = p'$.

$$\begin{tikzcd}
D' \arrow[dr, "w"] \arrow[drr, "p'"] \arrow[ddr, "q'"'] & & \\
& D \arrow[r, "p"] \arrow[d, "q"] & B \arrow[d, "f"] \\
& C \arrow[r, "g"'] & A
\end{tikzcd}$$

Figure 2: Pullback of two morphisms f and g

The generalisation of this operation to several objects and morphisms is called limit. There is also an intuition for limit in [5]:

“A diagram D in a category C can be seen as a system of constraints, and then a limit of D represents all possible solutions of the system.”

# 3 Modular specification of systems

Category theory has been used for a number of years as a framework for composing formal specifications based on early work by Goguen. Much of this work has concentrated on composition of algebraic specifications. More recently, Fiadeiro and Maibaum have developed an approach where each component of a system is described by a theory in temporal logic and theories are interconnected by specification morphisms [4]. Their approach has been adapted and applied [12, 7] and a tool has been implemented. In this section, we briefly present the main notions of this framework.

## 3.1 Category of specifications

We work in a particular category: the category of specifications with specifications as objects and specification morphisms as arrows. A specification is composed of two parts:

- the vocabulary needed to describe the component (we will call it extended signature in the following to distinguish it from the classical notion of (algebraic) signature), i.e. sorts, constants, attributes, actions;
- the behavior described by temporal logic axioms. In this paper we will not present a particular logic, as we will concentrate solely on the structural part (relations between the specifications). The framework has been applied to different linear temporal logics with actions [4, 12].

A specification is encapsulated by means of two notions: (extended) signature and locality. The signature is an initial means of delimiting the specification: each component has its own language (no global name space) and the logic of each component is parameterized by the signature. Moreover, each specification must respect the locality property. This property states that the attributes of the specification can only be modified by the actions of this specification. This property is essential to encapsulate each component and control the interactions between components.

A specification morphism $m : Spec1 \to Spec2$ associates each vocabulary element of $Spec1$ to a vocabulary element (of the same kind) of $Spec2$. The image of the axioms of $Spec1$ by the morphism must be true in $Spec2$.

Fiadeiro and Maibaum showed that, with these definitions of specification and specification morphisms, compositional verification can be achieved because the morphisms preserve all the properties. They also showed that the category of specifications was cocomplete (all the colimits exist) [4], and this is also proved by the implementation of this category in the tool (cf section 3.3 and appendix A).

## 3.2 Specification of systems.

The basic principle to specify a system using this framework is to specify each component of a system separately and then use the pushout (or colimit) to compose the specifications. A simple example will illustrate this. We wish to specify a system consisting of two components, B and C, each having a given behavior and one sending information to the other in a synchronous way. We first specify each component independently by means of attributes (that characterize the state of the component), actions the component can execute and axioms establishing constraints on these attributes and actions. Component B has an action $send(I)$ where $I$ is the type of information being sent; component C has an action $receive(Info)$. When both components are specified, we define a third specification (corresponding to $A$ in figure 1), which contains the elements that are shared by the two components. In the example, it contains a type $I$ and an action $com(I)$. We then define two morphisms, one that maps $I$ and $com(I)$ in A to $I$ and $send(I)$ in B, the other that maps $I$ and $com(I)$ in A to $Info$ and $receive(Info)$ in C. These sub-specification and morphisms allows to identify the sorts $I$ and $Info$ and the actions $send(I)$ and $receive(I)$. When we compute the pushout of this diagram, we get a specification describing the composed behavior of the two communicating components.

Based on this principle, Michel and Wiels have developed an extension of this approach where components are specified by modules with defined interfaces and several interconnection patterns are provided to compose the modules [7] (modules based on Ehrig and Mahr's work [3]). The aim of this framework is to simplify the user's task and provide some guidance. We will not detail this part here; however the extensions proposed in the following can be integrated in this framework.

## 3.3 Tool

To support this specification approach, a tool, Moka, has been developed [14]. It is implemented in layers. The first layer is a categorical kernel encoding in SML the notions of category theory that are necessary for this approach, essentially graphs, diagrams, categories, colimits, cocomplete categories, comma categories and adjunction. The categorical kernel was developed by J. Sauloy [10], following [9].

The second layer implements the notions of specification and module in an incremental way. Several different categories are built using the general constructions of the categorical kernel. These implementations are important, they answer two needs: find a construction process for each of the categories we need and prove that these categories are cocomplete (the construction process corresponds to the proof). Some details about the constructions are given in appendix A.

The third layer is an interface, in two parts: a language interface (external specification language and associated parser) and a graphical interface providing a friendlier access to the framework. The major part of the interface was developed by P. Michel.

Moka can be linked to a model checker or a theorem prover that is used to prove that the morphisms are correct and to verify some properties of the specifications.

# 4 Extensions to handle evolving specifications

Until now, we have used the previous framework with a "correct by construction" approach: components are specified, proved correct and then composed together in such a way to preserve their properties. However, it is desirable to be able to manage evolutions and to manage them at different levels (requirements, design, test) in a uniform way. We thus propose to extend the previous framework.

First, we present the basic principle that will allow us to handle partial specifications that are not completely correct or correctly related to each other. Then we explain how to represent different relationships between components at the same level of abstraction and present some operations that give information on the effects of these relationships on the system that is being built (without having to build it). We also show how to integrate the properties in the same framework and manage them in a uniform way. Finally, we study the capacity of this extended framework to deal with evolving specifications.

## 4.1 Morphisms and management of proof obligations

We saw in section 2 that a specification morphism $m : Spec1 = (Sig1, Ax1) \rightarrow Spec2 = (sig2, Ax2)$ must

- associate an element of $Sig2$ to each element of $Sig1$;
- preserve the axioms $Ax1$.

In the current version of the tool, each morphism is thus verified at the vocabulary level and at the behavior level. At the vocabulary level, the tool checks that each element of $Sig1$ has an image by the morphism, that its image is of the same kind (sort, constant, attribute, action) and has the same profile (type checking). At the behavior level, the tool just performs a simple check: it computes the difference between the sets of axioms $Ax2$ and $Ax1$ and generates proof obligations for all the axioms that are in $Ax1$ but not in $Ax2$. These proof obligations must be discharged for the morphism to be correct. It can be done by proving that these axioms are logical consequences of $Ax2$, using a model checker or a theorem prover.

To deal with incomplete specifications or morphisms, we need to relax the framework. The basic principle is that we allow morphisms that are not completely correct: the vocabulary part is well-formed but the proof obligations are not all discharged. It means that some inconsistencies may exist between the components. We can still compose the specifications and we store proof obligations associated to each morphism (and consequently to the target specification). This allows to deal with incomplete specifications and incomplete relationships between specifications.

In the current tool environment, we store the specifications, modules, and specification morphisms. We will have to add some bookkeeping information for each of these elements. This is discussed in section 6.

## 4.2 Relationships Between Specifications

In addition to morphisms that are not totally correct, we need to capture relationships between specifications for which there is no morphism linking them directly. Our approach here is to define sub-specifications that represent areas of overlap between specifications. There may be a number of such relationships between any two specifications. Rather than defining a single sub-specification to capture the total relationship, we define a separate sub-specification for each area of overlap. There are two advantages to this approach. First, it allows us to reason about interactions between the areas of overlap. Second, it helps us to maintain traceability because each area of overlap may have a different rationale, and may evolve at a different rate. We rely on the categorical framework to manage the proliferation of specifications that may result.

Consider two specifications $Sp1$ and $Sp2$ and two sub-specifications (for example $R$ and $R'$), each expressing a relationship between the two components. Each sub-specification captures an area of overlap between the two specifications.

$$\begin{tikzcd}
\mathrm{Sp1} & & & \mathrm{Sp2} \\
& \mathrm{R} \arrow[ul, "f1"] \arrow[urr, "f2"'] & \mathrm{R'} \arrow[ull, "g1"'] \arrow[ur, "g2"'] &
\end{tikzcd}$$

These sub-specifications link the two specifications at the vocabulary level. Defining an element $e$ in a sub-specification and associating it with $e1$ in $Sp1$ and $e2$ in $Sp2$ by means of two morphisms means that the two elements $e1$ and $e2$ will be identified in the system.

**Interaction between relationships.** We can use the limit of the diagram to compute some information about the effects of the combination of the different overlaps. The limit of the diagram is an object containing the couples $(e, e')$ such that $f1(e) = g1(e')$ and $f2(e) = g2(e')$, that is to say all the couples $(e, e')$ ($e \in R$ and $e' \in R'$) such that $e$ and $e'$ are associated to the same elements by the two relationships.

So this computation gives us the overlap of the overlaps.

**Effects of one relationship on the other.** We can also compute the pullback of $f1$ and $g1$. This gives the couples $(e, e')$ such that $f1(e) = g1(e')$. We can then compute the image of these couples by $(f2, g2)$. This gives all the couples of elements of $Sp2$ that would be identified (because of $R$) in the system. And reciprocally, we can compute the pullback of $f2$ and $g2$.

This computation allows us to detect "unexpected" interactions between relationships. An example of such an interaction is when the transitive closure of the set of identity relationships defined in $R$ and $R'$ results in an identity relationship between two elements of Sp2 that has not otherwise been captured explicitly.

*Remark*: in order to be able to compute the operations described previously, we have to prove that the category of extended signatures is finitely complete. This issue is discussed in section 6.

**Future work.** At present, we have only analyzed relationships at the vocabulary level (signatures), i.e. relationships between terminology of different specifications. Eventually we plan to study the effect of adding axioms to the sub-specifications to express consistency relationships, and hence formalize the notions presented in [2].

### 4.3 Properties

Once we have modeled relationships between specifications, we need to be able to represent properties of specifications, especially those properties that we wish to preserve as the specification evolves. It is important for the V&V process to be able to store these properties, but we do not want to embed them in the specifications for traceability reasons.

Let us consider the basic following case: a system is composed of two components specified respectively by specifications $Sp1$ and $Sp2$. These two components may share some elements declared in specification $Sp0$. The specification of the whole system (called $Sp12$) is in this case obtained by computing the pushout of $s01$ and $s02$.

$$\begin{tikzcd}
Sp0 \arrow[r, "s01"] \arrow[d, "s02"'] & Sp1 \arrow[d] \\
Sp2 \arrow[r] & Sp12
\end{tikzcd}$$

Properties can be associated to each specification. These are the properties that we expect the component to respect; that we need to prove on the component. We represent these properties in the same framework as the specifications and this allows us to use category theory and particularly categorical computations to manage them.

The user defines the properties and the specification they concern, for example property $ax1$ concerning $Sp1$. We then create a specification $P1$ with the same vocabulary as $Sp1$ and $ax1$ as axioms, and we define a morphism between $P1$ and $Sp1$. Hence we have a "shadow" subsystem of the system, storing its desired properties. For example, we would have for the system considered above the following scheme:

$$\begin{tikzcd}
 & & \mathrm{Sp12} & & \\
\mathrm{P1} \arrow[r, dotted, "ps1"'] & \mathrm{Sp1} \arrow[ur, dashed, "s112"] & & \mathrm{Sp2} \arrow[ul, dashed, "s212"'] & \mathrm{P2} \arrow[l, dotted, "ps2"] \\
 & & \mathrm{Sp0} \arrow[ul, "s01"] \arrow[ur, "s02"'] & & \\
 & & \mathrm{P0} \arrow[u, dotted, "ps0"'] & &
\end{tikzcd}$$

The advantage of this approach is that the management of properties and their status (proved, to be proved) is handled in a uniform way through the management of morphisms and proof obligations.

Moreover, we add morphisms between the different property specifications by projecting the morphisms of the system specification. On the example, we get:

$$\begin{tikzcd}
 & & \mathrm{Sp12} & & \\
\mathrm{P1} \arrow[r, dotted, "ps1"] & \mathrm{Sp1} \arrow[ur, dashed, "s112"] & & \mathrm{Sp2} \arrow[ul, dashed, "s212"'] & \mathrm{P2} \arrow[l, dotted, "ps2"'] \\
 & & \mathrm{Sp0} \arrow[ul, "s01"] \arrow[ur, "s02"'] & & \\
 & & \mathrm{P0} \arrow[u, dotted, "ps0"'] \arrow[uull, "p01"] \arrow[uurr, "p02"'] & &
\end{tikzcd}$$

This allows us to compute the properties associated to the specification $Sp12$: we just have to compute the pushout of $P1$ and $P2$ on $P0$. We get a property specification $P12$ and there is a morphism between $P12$ and $Sp12$.

$$\begin{tikzcd}
 & & & P12 \arrow[d, dotted, "ps12"] & & & \\
 & & & Sp12 & & & \\
P1 \arrow[r, dotted, "ps1"'] \arrow[uurrr, dashed, "p112"] & Sp1 \arrow[urr, dashed, "s112"] & & & & Sp2 \arrow[ull, dashed, "s212"'] & P2 \arrow[l, dotted, "ps2"] \arrow[uulll, dashed, "p212"'] \\
 & & & Sp0 \arrow[ull, "s01"] \arrow[urr, "s02"'] & & & \\
 & & & P0 \arrow[u, dotted, "ps0"] \arrow[uulll, "p01"] \arrow[uurrr, "p02"'] & & &
\end{tikzcd}$$

This morphism is obtained by applying the universal property of the pushout in the following way:

$P12$ is the pushout of $p01$ and $p02$. And we have $Sp12$, $s112 \circ ps1$ and $s212 \circ ps2$ such that $s112 \circ ps1 \circ p01 = s212 \circ ps2 \circ p02$ (*) so we know there exists a morphism $ps12 : P12 \to Sp12$ such that $s112 \circ ps1 = ps12 \circ p112$ and $s212 \circ ps2 = ps12 \circ p212$.

(*)Proof:
$s112 \circ ps1 \circ p01 = s112 \circ s01 \circ ps0$ (because of the way $p01$ and $s01$ are built)
$s112 \circ s01 \circ ps0 = s212 \circ s02 \circ ps0$ (property of the pushout $Sp12$)
$s212 \circ s02 \circ ps0 = s212 \circ ps2 \circ p02$ (because of the way $p02$ and $s02$ are built)

If no property is associated with one of the component specifications, we can still compute the pushout by taking as property specification a specification with the corresponding vocabulary but no axioms.

## 4.4 Management of evolving specifications

As pointed out in [8], dealing with changes is much easier with a structured specification. A well chosen structure helps to circumscribe the parts of the specification that must be modified more easily and to evaluate the consequences of the changes. We will now explain more precisely how this works in our framework.

The specification of a system is a diagram with a specification for each component, specifications and morphisms representing the relationships between the components; and in some cases, specifications and morphisms representing the properties of the components.

A change in the system may have different consequences. The first case is that the change corresponds to adding a new specification in the system. This does not induce any particular problems: thanks to the modularity of the framework, it is just similar to a step in the construction of the system specification.

A change can also result in modifying an existing specification or a relationship between specifications. In this case, we first identify the parts that need to be modified. The operations presented in 4.2 help in modifying the components and relationships and in understanding the impact of the changes. Then we check every morphism departing from or arriving at these specifications. We thus identify the consequences of the changes: if some related specifications need to be modified, this will be detected by the check on the morphisms. We also update the proof obligations attached to the morphisms: the changes may discharge some existing proof obligations, add new ones or necessitate re-proof of already discharged obligations. In that case, it would be useful to store information about already proved obligations and how they were proved (we discuss this in section 6). As the properties are integrated in the framework, they are updated in a uniform way.

# 5 Example

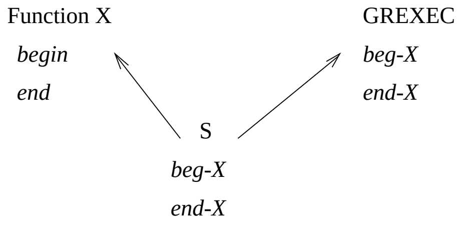
We have worked on a Space Shuttle change request as part of an earlier case study [15]. We think that this case study is a good testbed to explore the management of changes in large specifications. Here, we give a brief account of how our framework can be used on this case study, sufficient to illustrate the utility of the approach.

**Introduction.** As an operational vehicle, the Space Shuttle regularly needs updates to its flight software to support new capabilities (such as docking with the space station), replace obsolete technology (such as the move to GPS for navigation), or to correct anomalies. A change request typically consists of a selection of pages from current Computer Program Design Specification (CPDS) and Functional Subsystem Software Requirements (FSSR) specifications, with handwritten annotations showing new and changed requirements.

Our example is based on change request #90724, the East Coast Abort Landing (ECAL) automation CR. The change request covers changes needed to automate the entry guidance procedures for an emergency landing at sites on the East Coast or Bermuda, following a loss of thrust during launch, such that orbit cannot be attained. The core functionality of the change request covers the management of shuttle's energy during descent and the guidance needed to align it with the selected runway. Our approach is to model the old requirements in the FSSR first, and then update this model to reflect the changes listed in the CR.

The FSSR requirements are structured in functions. The main function GREXEC is executed at each time cycle and calls 13 other functions depending on the value of *iphase* (a variable representing the current guidance phase). We will not give here the detailed specification of the system and proofs of properties but we will explain why the extended framework is useful.

**Specification of the system.** We model each function using a separate specification. We will not present the bodies of these specifications here, but concentrate instead on the structure. We assume the bodies are specified in a suitable temporal logic. The sequencing can be represented in the following way: the GREXEC specification has an action *begin* and an action *end* for each of the 13 functions. Each function specification has its own *begin* and *end* actions. The action *begin_X* in GREXEC is identified to the action *begin* of the function X specification as follows:



The GREXEC specification describes the sequencing between functions and each function specification describes the corresponding behavior given in the requirements. There are of course other sub-specifications describing shared elements between specifications (e.g. shared variables). Thus we get a diagram specifying the different functions and how they are interconnected. The colimit of this diagram is the specification of the system.

**Properties.** Once we have modeled the specification structure, we want to check various properties of the system. For example, one of the variables computed by the system is the commanded roll angle *phic_at*. This angle is initialized in GRINIT and computed in two different functions TGPHIC and GRPHIC. One of the validation properties concerning the commanded roll angle *phic_at* is that the commanded angle is held at zero during the first two phases of entry (phases 5 and 6):

$$\Box(\mathit{iphase} = 5 \vee \mathit{iphase} = 6 \rightarrow \mathit{phic\_at} = 0)$$

Using the structure of the specifications, we can decompose this global property in several lemmas concerning different specifications:

- In GRINIT, we have to prove that the property $(\mathit{iphase} = 5 \vee \mathit{iphase} = 6 \rightarrow \mathit{phic\_at} = 0)$ is true at the initialisation.
- In GRPHIC, we have to prove that if *iphase* is not equal to 4, then $\mathit{phic\_at} = 0$.
- In GREXEC, we have to prove that

  * whenever GRPHIC is called, *iphase* can only be equal to 4, 5 or 6;

  * TGPHIC is not executed when *iphase* is equal to 5 or 6.

- The other functions do not modify *phic_at*.

Each local property is stored in a property specification and linked to the specification to which it relates. The local properties will be proved on the component specifications. The colimit of the diagram of property specifications then gives the properties that hold for the system.

**Change request.** The change request only modifies the requirements for GRPHIC. So we know we only have to change the GRPHIC specification and reconsider the properties attached to this specification.

The structure of the specification helps a lot in identifying the part of the specification that must be modified. The fact that the properties are integrated in the same framework allows us to update the status of these properties (this can be automatically computed by checking the morphisms) and thus know which parts need to be proved or re-proved.

**Conclusions.** Our original approach to this case study was to build a single, large formal model for the entry guidance requirements, validate this model against the properties, update the model according to the change request and then re-validate the properties. There were significant problems in doing this due to a loss of traceability between the change request and the formal model. In particular, there were some aspects of the change request that could not be validated as they had no correspondence in the formal model.

The framework we have described in this paper offers a number of advantages for this case study. The structure of our formal specifications faithfully reflects the structure of the documented requirements. This improves our ability to trace between the two, and ensures that we accurately capture the change request in our model. It also improves readability of the formal specification. The structure then allows us to isolate changes, and reason about their impact. At the property level, the framework allows compositional verification: global properties are decomposed following the structure of the specification. The integration of properties in the same framework allows us to manage them in a uniform way.

# 6 Implementation

To support this framework, two kinds of extensions are needed to the tool described in section 3.3. First, we need to be able to compute pullbacks and limits of signatures. Then we have to add mechanisms to store and manage the proof obligations.

**Finite completeness of the category of extended signatures.** The problem is that the category of extended signatures as it is built in the tool is not finitely complete (cf appendix A and [11]). It has the products and equalizers and thus the pullbacks, but no terminal object.

We have identified two solutions to this problem. The first solution is to define and implement a slightly different category of signatures that is finitely complete [11]. This solution is general but necessitates significant updates of the existing implementations.

The second solution is to consider only the vocabulary elements (constants, attributes, actions) without their profiles. In that case, we only need to compute limits in the category of finite sets, which is a finitely complete category (cf appendix A for details). This ad hoc solution is sufficient for the computations described in section 3.2. Indeed, we do not need to build limits as objects of the category of specifications, we only use them to give additional information to the user.

**Management of the proof obligations.** To support the use of partially correct morphisms and the management of change, we need to add information about the elements of the environment. For each morphism, we need to store the following information:

- nature: user defined morphism or automatically generated morphism; morphism of the system, between properties and system or between properties;
- proof obligations attached to this morphism with their status (proved, to be proved).

For each specification, we need to store the following information:

- nature: property or system; user defined or computed. Additionally, for computed specifications, the operation and its arguments;
- morphisms from and to this specification.

This additional information will be managed at the interface level in the tool. These are essentially bookkeeping operations that support the user but do not require further modification at the category-theoretic level.

# 7 Conclusions and future work

We have presented in this paper several extensions to an existing categorical framework. Category theory provides an excellent basis for providing structure in formal specifications. It provides a coherent and well-founded theoretical basis for representing structure in existing specification languages, thus avoiding the need to add structuring primitives within each language.

The extensions described in this paper were motivated by the need to support the evolution of large specifications. For large specifications evolved by a team of people, a “correct by construction” approach does not suffice. Hence, we needed to consider how to adapt the categorical framework to support the following requirements:

- Support for traceability as a specification evolves, by explicitly representing relationships between specification components, and between specification and validation properties. The framework needs to support an ability to trace these relationships to their rationales, and to support tracing of the impacts of change.

- Support for compositional verification, so that global system properties can be decomposed across the structure of a specification, and such that we limit the number of proofs that have to be re-checked when a change is made.

- Support for the process of defining relationships (morphisms) between specification, with the ability to handle morphisms that are only partially correct.

- An approach that adapts well to legacy systems, where changes to an existing specification need to be verified.

To meet these requirements, we first adapted the categorical framework to permit morphisms that are well-formed but not totally correct, and provided bookkeeping support for keeping track of the proof obligations arising from each morphism. We then provided a framework for capturing relationships between specification components by representing areas of overlap as sub-specifications. By using a separate subspecification for each area of overlap, we can reason about interaction between these relationships, and can preserve traceability as the specification evolves. Finally, we incorporated specification properties into the same framework, so that the relationships between properties and specifications could be managed using the same approach.

We demonstrated how the framework might apply to a case study of the verification of a change request for the Space Shuttle. The approach should greatly improve our ability to perform verification and validation of change requests, as it allows us to construct a model that more accurately reflects the existing structure of the specification, and to isolate the impacts of the change using this structure.

The application to the case study is still in progress, we need to implement the proposed extensions in order to use them for the management of change. We also plan to further study the representation of different relationships between components and their interaction as suggested in section 4.2.

Long term areas of future work include the use of different specification formalisms and eventually of heterogeneous specifications that allow us to better describe different parts of a system. We also wish to use another existing categorical framework, Specware [13], and compare it to our approach. Finally, we are interested in the integration of test cases in the same framework [1].

# References

[1] M. Doche, C. Seguin, and V. Wiels. Generation of functional test cases from modular specifications. Submitted.

[2] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering journal*, 11:31–43, 1996.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990. Modules specifications and constraints.

[4] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[5] J.A. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Science*, 1(1), March 1991.

[6] J.A. Goguen and R.M. Burstall. Some fundamental algebraic tools for the semantics of computation. part1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science*, 31(1,2), May 1984.

[7] P. Michel and V. Wiels. A Framework for Modular Formal Specification and Verification. In *Proceedings of FME'97*, number 1313 in LNCS. Springer-Verlag, 1997.

[8] S. Miller and K. Hoech. Specifying the mode logic of a flight guidance system in CoRE. In *Proceedings of FMSP 98, Formal Methods in Software Practice*, 1998.

[9] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* International Series in Computer Science. Prentice Hall, 1988.

[10] J. Sauloy. Interconnexion de Modules. Internal ESF Research Report, Centre d'tudes et de Recherches de Toulouse, 2 av. Edouard Belin, B.P. 4025, 31055 Toulouse Cedex, September 1992.

[11] J. Sauloy and V. Wiels. Finite completeness of the category of signatures. Forthcoming.

[12] C. Seguin and V. Wiels. Using a Logical and Categorical Approach for the Validation of Fault-tolerant Systems. In *Proceedings of FME'96*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[13] Y.V. Srinivas and R. Jüllig. Specware: formal support for composing software. In *Proceedings of the conference on Mathematics for Program Construction*, 1995.

[14] V. Wiels. *Modularité pour la conception et la validation formelles de systèmes.* PhD thesis, ENSAE, 1997.

[15] V. Wiels and S. Easterbrook. Formal modeling of space shuttle software change requests using SCR. Forthcoming.

[16] M. Wirsing. Algebraic specification. In *Handbook of theoretical computer science, Formal Models and Semantics*, volume B, pages 675–788. Elsevier and MIT Press, 1990.

# A Details on the categorical constructions

The second layer of the tool implements several categories: the category of finite sets, the category of algebraic signatures, the category of extended signatures, the category of specifications, the category of parameterized specifications and the category of modules. And all these categories are built as cocomplete categories.

We only give here details about the category of extended signatures. Its construction process is inspired by the encoding of algebraic signatures as comma categories [6].

**Com3 category.** The objects of the com3 category $(L, R, S, T)$ are triples $((a, u, b), (d, v), (e, w))$ where $L : A \to C$, $R : B \to C$, $S : D \to C$, $T : E \to C$, $a$ is an object of $A$, $b$ is an object of $B$, $d$ is an object of $D$, $e$ is an object of $E$, $u : L(a) \to R(b)$, $v : S(d) \to R(b)$, $w : T(e) \to R(b)$ are morphisms of $C$. Morphisms of $(L, R, S, T)$ category are $(f, g, h, i)$ where $f : a_1 \to a_2$ is a morphism of $A$, $g : b_1 \to b_2$ is a morphism of $B$, $h : d_1 \to d_2$ is a morphism of $D$, $i : e_1 \to e_2$ is a morphism of $E$ such that the three squares commute:

$$\begin{tikzcd}
L(a_1) \arrow[r, "u_1"] \arrow[d, swap, "L(f)"] & R(b_1) \arrow[d, "R(g)"] & S(d_1) \arrow[r, "v_1"] \arrow[d, swap, "S(h)"] & R(b_1) \arrow[d, "R(g)"] & T(e_1) \arrow[r, "w_1"] \arrow[d, swap, "T(i)"] & R(b_1) \arrow[d, "R(g)"] \\
L(a_2) \arrow[r, swap, "u_2"] & R(b_2) & S(d_2) \arrow[r, swap, "v_2"] & R(b_2) & T(e_2) \arrow[r, swap, "w_2"] & R(b_2)
\end{tikzcd}$$

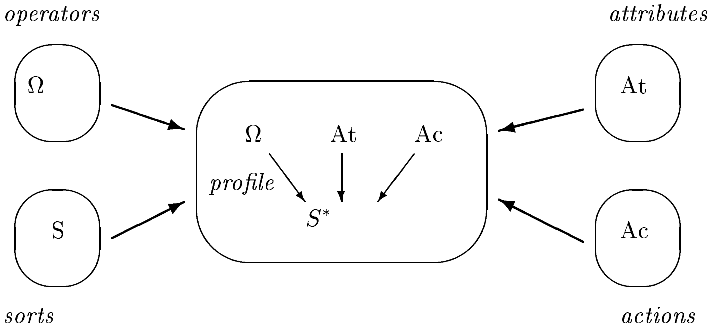
**Cocompleteness.** If $L$, $S$ and $T$ are cocontinuous functors and if $A, B, C, D$ are cocomplete categories, then $(L, R, S, T)$ is cocomplete.

The proof of this theorem is given in [14].

**Completeness.** If $R$ is a continuous functor and if $A, B, C, D$ are complete categories, then $(L, R, S, T)$ is complete.

The proof of this theorem is given in [11].

**Category of extended signatures.** The category of extended signatures (with attributes and actions) is the com3 category $(SetToInf, Star, SetToInf, SetToInf)$ where $SetToInf : FinSet \rightarrow InfSet$ transforms the finite set $S$ into the same set $S$ but viewed as an infinite set, and $Star : FinSet \rightarrow InfSet$ associates to a set $S$ the free monoid $S^*$.



The cocompleteness of the category of extended signatures results from the cocompleteness of the category of finite sets and the cocontinuity of the $SetToInf$ functor (which are proved in [14]).

**Completeness.** The category of extended signature is not complete (the $Star$ functor is not continuous). We can compute products and equalizers in this category and thus pullbacks, but there is no terminal object [11]. We can define another category of enumerables extended signatures that is finitely complete as explained in [11] or adopt the following ad hoc solution.

We can consider only part of the specifications: a specification is an object $(((Op, u, S^*), (At, v), (Ac, w)), Ax)$, we project this to get an object $(Op, At, Ac)$ containing only the operators, attributes and actions of the specification. We also project the specification morphisms $(f, g, h, i)$ in $(f, h, i)$. We can then compute limits for each kind of element (operator, attributes, actions) in the category of finite sets, which is finitely complete.