Towards a categorical framework to ensure correct software evolutions

Sylvain Bouveret, Julien Brunel, David Chemouil, Fabien Dagnat

TOWARDS A CATEGORICAL FRAMEWORK TO ENSURE CORRECT SOFTWARE EVOLUTIONS

SYLVAIN BOUVERET, JULIEN BRUNEL, DAVID CHEMOUIL, AND FABIEN DAGNAT

Originally published in *Workshop on Hot Topics in Software Upgrades (HoTSwUp 2011), Proc. IEEE 27th International Conference on Data Engineering Workshops*, 2011.
DOI: 10.1109/ICDEW.2011.5767625.

Abstract. Distributed software, such as satellite software are now developed and managed by several actors. In this context supporting the maintenance and therefore the evolution of such applications is complex and need a formal framework. In this article, we propose a first step towards such a formal framework to ensure the correctness of software evolutions. Using category theory, we can model software and represent patches. This modeling allows to identify the proof obligations that the provider of a patch has to discharge in order to ensure that its patch preserves the correctness of the software.

1. Introduction

A satellite is a spacecraft aimed at executing a mission such as scientific observations. To perform its mission, a satellite is controlled by its *flight software* while the *payload software* is in charge of the mission specific operations (*e.g.* taking photos). The flight software, having to guarantee that the satellite assumes its mission, is a highly critical software. Once the satellite has been launched, its flight software must be updated to fix bugs, to adapt to mission evolutions or to hardware degradation (mainly due to cosmic particles). This maintenance process is a critical craft using patches and requiring a high level of expertise. Indeed, while the process and the quality assurance for producing the initial software is highly codified, the patching process itself is rarely formalised.

The current practice consists mainly on producing by-hand the new source code. Furthermore, as the link between the ground stations and the satellite has a low bandwidth assigned to patches, they must be the smallest possible. This implies, for example, that we must avoid to move data and code in the memory. While this practice is acceptable for simple (local) patches, producing a more complex patch and ensuring its correctness is hard. Worse, this patch process does not scale to the management of a constellation of satellites because a patch is specific to a satellite[1]. In a previous paper [1], we proposed to replace it by an MDE approach where patches would be obtained by model transformation and code generation. In this paper, we would like to focus on the need for formal mathematical foundations of patches. This paper synthesises ideas developed by the authors during projects and collaborations with the major European satellite manufacturers such as for example the SPaCIFY project[2].

2. Satellite as distributed systems

Patching a satellite is a difficult task requiring expertise. While better production processes may lower the cost of producing a patch, the current practice of hand-written patches works well. But two major evolutions foreseen for future satellites call into question the confidence in this process:

(1) future satellite software architecture will be distributed. Both because the platform will include several processors and because of the massive use of virtualisation.

[1]For example, the memory mapping is unique because the damages memory banks suffer are different on each satellite.
[2]http://spacify.gforge.enseeiht.fr

(2) future mission will be more complex and will require the cooperation of a group a satellites called a constellation. It may be a *static* constellation meaning that the satellites have been developed specifically to cooperate. But studies are now conducted on how to *dynamically* build a constellation by asking a group of already launched and separately developed satellite to cooperate.

These evolutions will require that the flight software becomes a distributed software. Furthermore, to control a constellation a flight software will be distributed over several spacecraft and may also include software components executing on ground stations. To simplify the presentation we assume that a flight software S is a (distributed) configuration of *architectural elements* $\{E_i\}$. Architectural elements may either be component or connector instances. By component we simply mean a unit of computation that communicates with other components through connectors. A link between two elements is called a *wire* (there may be many wires between two elements).

The major difficulty when managing patches of such systems is the guarantee of the required high level of safety. Indeed, in the spatial context, each satellite or software may be specified, developed or maintained by a different producer. For a given satellite, one of the contractors may be granted the responsibility to manage and validate the patches. But, in a more open context, where a constellation may depend on the cooperation of satellites owned by various competing entity, it is more difficult to have a unique manager. Our aim is to provide a mathematical framework that will help collecting all the proof obligations required for the application of a patch.

Going back to our model, each architectural element E_i may be specified, developed or maintained by a different entity. *Patching* the system $S = \{E_i\}$ cannot follow the usual version management approach where all the application is managed by a single entity. To be able to manage patches in our fragmented context, one needs a way to compose a *global patch* P over S out of *local patches* $\{p_i\}$. Intuitively, our idea here is to use the system architecture as a guide for this composition. Each patch p_i relates to an architectural element E_i. This differs from the usual practice of *Revision Control System* (such as subversion or Darcs) where a patch is composed of changes made to files. Work has been made to propose an algebra of file patches [2, 3]. But they focus on the composition and commutation of changes. While this contribution is valuable, it does not address the problems raised by distributed systems patching.

Design and implementation of such patches can then follow the usual process. Managing (designing, implementing, validating, applying, canceling) patches to S is reduced to the management of the local patches $\{p_i\}$ and their dependencies. When following this approach, the difficulty is to control the effect of a local patch on the whole architecture to be able to prove the safety.

We believe that category theory is well-suited both for representing local patches and their effects on the whole architecture, at least for four reasons, also advocated in [4]. The first one is that category theory is the perfect tool for representing and reasoning about architectures, viewed as objects and relations between them. The second reason is that category theory is abstract enough to stay independent of any specification language, while providing the useful composition primitives to the specification language. The third reason is that, as we shall see later, the correctness of any construction is guaranteed "automatically" by the framework itself, as soon as some basic properties are verified. Finally, representing patches as functors, like we do in our framework, ensures that the composition of two correct patches yields a correct patch.

The contribution of this paper is a framework that formalises the semantics of a (locally) patched architecture. More precisely, we propose a categorical framework where software elements are composed of a contract and a body. An architecture is correct if, for each element, the body satisfies the contract. In this framework, we give a sufficient condition that a patch must ensure to maintain the correctness property of an architecture. We would like to emphasize the importance of morphisms. The morphism between a body and the associated contract explains why the body satisfies the contract. The proof obligations needed to ensure the correctness of any patch causing an evolution of a body result from

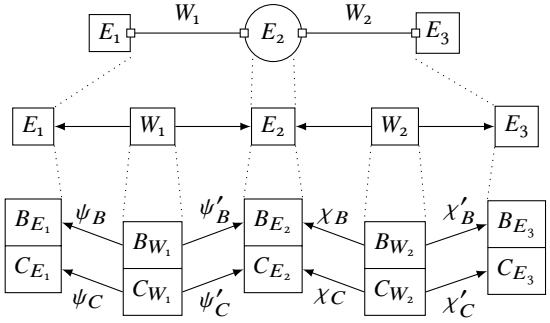


FIGURE 1. A framework for architecture description: from architecture description (upper level) to categories (lower level).

these morphisms. So the essence of the proposal is not the pair contract / body but the morphisms and therefore the categorical structure.

3. A Basic Categorical Framework

3.1. **Architecture Description.** We aim at defining a framework that does not make too many hypotheses over the evolving system. Thus, we only consider that a system is described as a configuration of *elements* (component and connector instances). Taking inspiration from [4], we require elements to be described in two parts. The first one is the *body* while the second one consists of a *contract* asserted over the former (possibly in the sense of [5]). This dichotomy enables to track the impact of a body change over a contract and to detect that a body has not been updated with respect to its associated contract. As we strive for a notion of correct evolution, we consider that in the initial configuration, the contract asserted over a body of any element is verified.

Fig. 1 sums up the setting to be introduced. In the upper level, a connector E_2 links two components E_1 and E_3 through the wires W_1 and W_2. In the middle level elements and wires are mapped to a same kind of objects linked by arrows (the morphisms of a suitable category **Elem** described later). Finally, the lower level gives a detailed view of objects and morphisms: objects comprise a body and a contract, and arrows between objects are actually pairs of morphisms.

3.2. Formalisation.

3.2.1. *Basic Notions.* Our treatment of evolution is independent from any "common" logic (propositional, first-order, Floyd-Hoare...). In fact, the formalism is not even required to be a logic. However, we will illustrate the framework on *linear temporal logic* (LTL) [6] to simplify the presentation.

A formalism comes with a syntax, *i.e.* a set of well-formed *sentences*. These could be for instance formulae representing axioms and theorems. Sentences are built inductively out of a *signature* that declares the (possibly typed) vocabulary used in the formalisation. For instance, a set of sentences in LTL is built using the usual Boolean ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) and temporal (G for "always", X for "next") connectives and a signature (base vocabulary) of proposition names. Now, a signature morphism is a map that renames and/or merges sort and operation symbols (while preserving the correct "typing", if any). Then a signature morphism φ can be extended inductively to a *morphism of sentences* $\hat{\varphi}$. It translates a sentence built over a signature Σ into one over a signature Σ', by renaming and/or merging symbols while preserving the sorting of symbols.

As usual in formal systems, we postulate that the *semantics* of a sentence is described using *models*. We write $\mathcal{M} \models E$ when a model $\mathcal{M}$ satisfies the sentences in E[3]. In a concrete setting, a model is for

[3]Note that, usually, the semantics of sentences is built from the semantics of signatures but we do not enter into details here.

instance a program implementing the constraints imposed by E. A sentence s is the consequence of a set of sentences E, written $E \models s$, if every model of E is also a model of s ($\mathcal{M} \models E \Rightarrow \mathcal{M} \models s$). In LTL, a model is a transition system. One may then state that $G(a \wedge b) \models G(a \vee b)$, because in every LTL model where $a \wedge b$ is true, $a \vee b$ is true.

3.2.2. *The Categorical Framework of Presentation.* We can now introduce the formalisation of bodies and contracts.

Definition 1 (Presentation)**.** *A* (theory) presentation $P = (\Sigma, E)$ *consists of a signature* Σ *and a set* E *of sentences over* Σ*; and the associated presented theory is the pair* (Σ, E') *where* E' *is the closure of* E *w.r.t. semantic consequence.*

As usual, the closure is defined as the least fixed point of the function $f_{\models} : F \mapsto \{s \mid F \models s\}$, which exists, from the fixpoint theorem. In our framework, both the body and the contract of elements are described as finite presentations.

Definition 2 (Presentation Morphism)**.** *A* presentation morphism $\varphi : (\Sigma, E) \to (\Sigma', E')$ *is a signature morphism* φ *from* Σ *to* Σ' *s.t. for its extension* $\hat{\varphi}$*, we have* $\hat{\varphi}(E) \subseteq E'$*.*

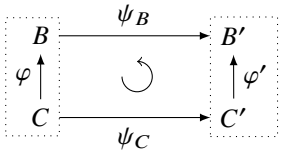
Informally, a presentation morphism preserves the truth from one presentation to another, while allowing to merge and rename symbols. Here, all the properties in the contract of an element are consequences of its body. So there is a presentation morphism from the contract to the body. Intuitively, a presentation morphism explains how its source may be embedded as a "part" of the target. Let us stress that each presentation has its own local vocabulary. Renaming and merging might not seem interesting but they ensure a form of decoupling between the contract and the body. For instance, a contract can provide several different properties that, under the hood, reduce to a single service.

Presentations and their morphisms form a category, called **Pres** where every finite diagram admits a colimit [7]. Intuitively, it is always possible to "coalesce" (using a standard algorithm) a finite set of presentations related by morphisms into one large presentation modulo renaming and/or merging while preserving "truth" and the "well-typing" of sentences.

3.2.3. *A Framework for Architectural Elements.* We can now define our framework for architectural elements.

Definition 3 (Element)**.** *An* (architectural) element $E = (C, \varphi, B)$ *is given by two presentations* C *(the contract) and* B *(the body) and a presentation morphism* $\varphi : C \to B$*.*

Definition 4 (Element Morphism)**.** *A morphism* ψ *from an element* $E = (C, \varphi, B)$ *to an element* $E' = (C', \varphi', B')$ *is a pair* (ψ_C, ψ_B) *of presentation morphisms* $\psi_C : C \to C'$ *and* $\psi_B : B \to B'$ *s.t.* $\varphi; \psi_B = \psi_C; \varphi'$*,* i.e. *s.t. the following diagram commutes:*



This condition means the way contract axioms map to body theorems is "preserved" by the element morphism.

Proposition 1. *Architectural elements and their morphisms form a category, called* **Elem***, that admits all finite colimits.*

This result follows from the fact that **Elem** is a so-called *comma* category, namely $(Id_{\mathbf{Pres}} \downarrow Id_{\mathbf{Pres}})$.

Definition 5 (Architecture)**.** *An architecture* $\mathcal{A}$ *is a finite subcategory of* **Elem** *closed under colimits.*

An architecture model can now be interpreted as follows: (1) every component or connector is interpreted as an object in the category **Elem**; (2) every wire gives rise to an object in **Elem** and a pair of morphisms identifying names on both ends of the wire. Apart from this, a wire does not convey any meaning, so its contract and body are empty. Then the semantics of the architecture model is given by the colimit of its interpretation, which exists by Prop. 1. Fig. 2 shows the colimit C of elements E_1, E_2, and E_3 connected through the wires W_1 and W_2. The dotted arrow is *the unique morphism* from C to C' and roughly means that C is unique up to isomorphism and the smallest object out of all "candidates" (such as C').

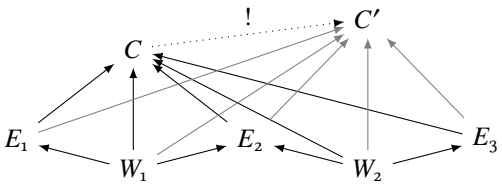


FIGURE 2. Five elements and their colimit C in category **Elem**.

Let us stress that conforming to our categorical framework imposes to discharge some proof obligations stating that:

(1) the signature map between a contract and a body is a morphism, *i.e.* contract axioms maps to body theorems;
(2) the signature maps between a wire and its ends E and E' are morphisms, *i.e.* each symbol in the wire is mapped to one symbol in E and one in E'.

4. PATCHES

4.1. **Semantics of Patches.** This section defines a patch as an arbitrary system modification. Among all patches, we focus on those preserving certain architectural properties and call them *transformations*. Our preliminary framework is currently limited, but we believe it lays the foundations for a comprehensive semantics for transformations. In particular, some proof obligations must be checked to ensure that a patch is a transformation. They come in addition to the proof obligations over the architecture itself.

The only patches considered in this paper are as follows:

P1) renaming a proposition inside a contract/body (with a potential impact on the associated wires)
P2) removal/addition of axioms inside a contract/body
P3) modification of a wire leaving the connected elements unchanged
P4) combination of several elements of the same nature into one (taking the wiring into account)
P5) any composition of the former elementary patches

In all these cases, the existence of links between elements must be preserved. In our framework, this is done by requiring transformations to be *functors*. A functor F from an architecture $\mathcal{A}_1$ to an architecture $\mathcal{A}_2$ is a pair of maps (F_e, F_m) s.t.

- F_e maps each element of $|\mathcal{A}_1|$ to an element of $|\mathcal{A}_2|$;
- F_m maps morphisms of $\mathcal{A}_1$ to $\mathcal{A}_2$ ($\forall \psi : E \to E'$ in $\mathcal{A}_1$, there is $F_m(\psi) : F_e(E) \to F_e(E')$ in $\mathcal{A}_2$) preserving identity morphisms and the composition of morphisms.

Definition 6 (Transformation)**.** *A transformation is a functor of architectures that preserves colimits.*

For each of the aforementioned classes of patches, we list below the conditions under which they are transformations. That is, we describe[4] the proof obligations to be discharged.

[4]We do not give the proof in this paper that these conditions are sufficient to define a transformation.

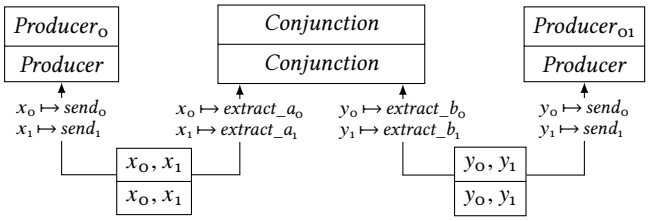


FIGURE 3. The example architecture.

P1) This is always a transformation provided that the morphisms that involve the modified atomic proposition are updated. We possibly have to update both presentation morphisms (inside the considered element) and element morphisms (relating the element to other elements).

P2) First note that all morphisms that come from a wire are not impacted by such patches.

The removal of a contract axiom is always a transformation, since the presentation morphism from the contract to the body is obviously preserved. However, the addition of a contract axiom is not necessarily a transformation. In order to preserve the morphism between the contract and the body, we must prove that the translation (according to the signature morphism) of the new contract axiom is a theorem according to the body presentation.

The removal/addition of a body axiom works dually. The addition of an axiom is always a transformation, and the removal only if we establish that contract axioms are still translated into theorems in the (new) body.

P3) This is always a transformation provided that the new wiring still defines a signature morphism.

P4) This is a transformation when the wires that were connected to the combined elements are updated appropriately. Each wire connected to these elements before the patch is connected to the element resulting from the combination after the patch. Note that if some propositions of elements to be combined have the same name, there are two options: either merging them, or renaming them to get a disjoint union. Both options define a transformation, but the semantics of the resulting architecture may differ.

Note that, as already stated, our definition excludes some patches, the most notable being the addition of elements or wires. A larger class of patches will be addressed in an extension of our framework involving more complex constructions than mere functors. Let us also insist on the fact that our approach is independent from the inner nature of elements (given by the category **Pres**). Indeed, we only require body and contracts to be objects in a category with all finite colimits.

4.2. **Illustration on a Simple Producer-Consumer Example.** We illustrate our work with a simple Producer-Consumer example inspired by [8] and presented in Fig. 3.

4.2.1. *Example Description.* Two bit producers send bits to a connector that computes and outputs their conjunction. (We do not describe the Consumer in the following.) These elements are built from the presentations described in LTL in Fig. 4.

Fig. 4 (left) shows the contract of a producer that successively produces and sends bits. The atomic proposition $prod_0$ (resp. $prod_1$) encodes the production of 0 (resp. 1) and *current* encodes the value of the last produced bit (true for 1, false for 0). Similarly, when $send_0$ (resp. $send_1$) is true, a 0 (resp. 1) is sent. Fig. 4 (middle top) contains the body $Producer_0$, of a producer that cannot produce 1. He can only produce 0, send 0, or leave everything unchanged. Fig. 4 (middle bottom) shows the body of a producer that cannot produce two equal bits consecutively. After producing 0, the next bit must be 1 (if it produces one) and after 1 must come 0.

From these three presentations, we define the two elements representing producers. Both share the contract defined by the presentation *Producer*. The presentation $Producer_0$ (resp. $Producer_{01}$) is the body

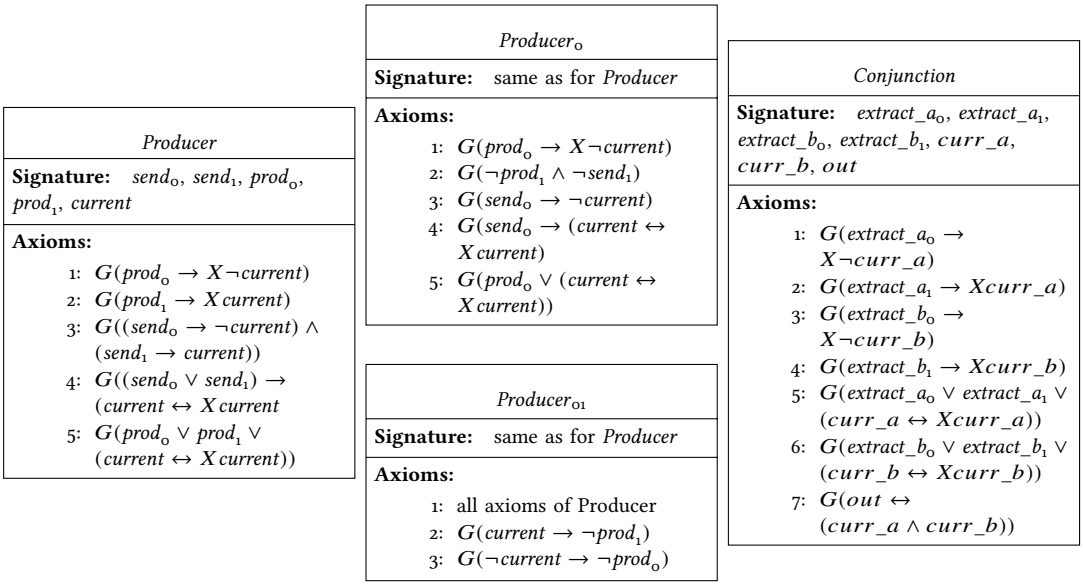


FIGURE 4. Body and contracts in the example.

of the producer represented at the left (resp. right) of Fig. 3. The fact that both bodies respect their contracts is ensured by the existence of two morphisms one from *Producer* to $Producer_0$, and one from *Producer* to $Producer_{01}$. Such morphisms are based on identity signature morphisms (since contracts and bodies share the same signature) and the fact that all axioms of *Producer* are theorems in the theories presented by $Producer_0$ and $Producer_{01}$.

We also define a simple conjunction connector in Fig. 4 (right). The body and the contract of this connector are the same. This connector computes the conjunction of the two bits received from the producers. It may send

4.2.2. *Producer-Consumer Patches.* We are now able to consider some of the aforementioned patches on this example.

- Renaming of a proposition: Let us change the name of the proposition *current* to *buffer* in the $Producer_0$ body. This modification implies the update of the presentation morphism that relates the body to the contract in order to have a correct patch. The proposition *current* of the contract *Producer* must be mapped to the new proposition *buffer* of the body $Producer_0$ so that the old guy body still implements its contract.

  If the renaming occurs on a proposition interfering with other elements (such as $prod_0$) then modifications on wires must be considered in order to have a correct patch.
- Removal/addition of a contract axiom: Consider the removal of an axiom in one of the producers' contract. As stated in section 4.1, this is a transformation, since the body still respects the new contract. Note that the contract of the whole system (defined as the architecture colimit) is impacted by this transformation.

  Consider now a patch defined by the addition of an axiom to the contract of the producer on the left of Fig. 3. We define a new specification *Producer_bis* by adding the axiom $Gprod_0$ to *Producer*. We must prove that we still have a morphism between the new contract *Producer_bis* and the body $Producer_0$. This amounts to check that the translation of the new axiom according to the signature map (the identity in this case) is a theorem according to the theory presented

by the body $Producer_0$. Here $G\neg prod_0$ is not provable from the axioms of $Producer_0$. Thus, this patch is not a transformation.

- Removal/addition of a body axiom: We now remove axiom 2 of $Producer_{01}$. The new body still respects its contract (all axioms of $Producer$ are still axioms, and thus theorems of the new body). So this patch is a transformation. Note that it modifies the body of the whole system, since it is now possible for the producer on the right of Fig. 3 to produce several 1 successively.
- Combination of several elements of the same nature: Let us consider the combination of both producers into a single one. We can define several patches, depending on the choices concerning the merging of the propositions that have the same name. If we merge all of them, the resulting element body produces only one bit at each instant. This bit must comply with both the axioms of $Producer_0$ and of $Producer_{01}$. It is sent twice to the connector $Conjuction$, one with each wire. The only possible behaviours compatible with $Producer_0$ and $Producer_{01}$, are to never produce anything, or to produce a 0 and then no more bit.

  If we do not merge any proposition, the resulting element body produces two bits at the same time: one according to the body axioms of $Produce_0$ sent through one wire, the other according to the axioms of $Produce_{01}$ sent through the other wire. The global behaviour is actually the same as before the patch. As stated in section 4.1, both patches define a transformation.

5. Conclusion

In this paper we propose a first step towards a categorical formalization of architectural patches. We have introduced a generic categorical framework for specifying and representing architectures. The semantics of component aggregation is given, as in prominent previous approaches, by the colimit of a diagram. In this context, a patch is a functor between two architectures. With this formalization, we are not only able to assert the correctness of a given patch (which translates *e.g.* to proof obligations), but the semantics of the component composition also implies that the resulting architecture is correct. The precise form of the proof obligations and the amount of work required to discharge them must be explored by looking at more complex and realistic examples (such as a satellite flight software).

The main limitation of this proposal is that the model, in its current state, is not able to deal with complex architectural evolutions, such as adding or removing elements. We believe that managing such kind of evolutions is possible in our framework, but may involve more complex categorical concepts. Another natural extension of our framework would be to define contracts not only for elements but also to groups of elements or more globally to the whole architecture. This may require the proposal of a more complex representation of what is an architecture in the categorical framework. We leave all these questions for future work.

On a longer-term scale, we intend to explore the decomposition of patches. Indeed, specifying (functional as well as non-functional) evolutions for a distributed system necessitates a formal approach to decompose this (global) evolution into evolutions of the architectural elements. Each (local) evolution is then designed and results in a (local) patch. Then, these patches are validated and recomposed. Having a formal framework to support this process would enable to safely establish the list of proof obligations all parties have to discharge.

Acknowledgment

We thank Virginie Wiels for stimulating discussions about earlier work [4] on categorical approaches to evolution.

References

[1] J. Buisson, C. Carro, and F. Dagnat, "Issues in applying a model driven approach to reconfigurations of satellite software," in *Workshop on Hot Topics in Software Upgrades*, Nashville, Tennessee, USA, Oct. 2008.
[2] I. Lynagh, "An algebra of patches," 2006, http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf. A more recent version of the framework may be found at `http://projects.haskell.org/camp/`.
[3] J. Jacobson, "A formalization of darcs patch theory using inverse semigroups," UCLA, Tech. Rep. 09-83, 2009.
[4] V. Wiels and S. M. Easterbrook, "Management of evolving specifications using category theory," in *ASE*, 1998, pp. 12–21.
[5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999.
[6] A. Pnueli, "The temporal semantics of concurrent programs," *Theoretical Computer Science*, vol. 13, pp. 45–60, 1981.
[7] J. L. Fiadeiro and T. Maibaum, "Temporal theories as modularisation units for concurrent system specifications," *Formal Aspects of Computing Journal*, vol. 4, no. 3, pp. 239–272, 1992.
[8] P. Castro, N. Aguirre, C. López Pombo, and T. Maibaum, "Towards managing dynamic reconfiguration of software systems in a categorical setting," in *ICTAC 2010*, 2010, pp. 306–321.

ONERA-DTIM, 2, avenue Édouard Belin, BP74025,, 31055 Toulouse Cedex 4, France
E-mail address: `firstname.lastname@onera.fr`

ONERA-DTIM, 2, avenue Édouard Belin, BP74025,, 31055 Toulouse Cedex 4, France
E-mail address: `firstname.lastname@onera.fr`

ONERA-DTIM, 2, avenue Édouard Belin, BP74025,, 31055 Toulouse Cedex 4, France
E-mail address: `firstname.lastname@onera.fr`

Institut TELECOM / TELECOM Bretagne, Université Européenne de Bretagne, Technopôle Brest-Iroise - CS 83818, 29238 Brest Cedex 3, France
E-mail address: `fabien.dagnat@telecom-bretagne.eu`