Formal Testing of Correspondence Carrying Software

Marius C. Bujorianu, Manuela L. Bujorianu\textsuperscript{1}, Savi Maharaj\textsuperscript{2}

\textsuperscript{1}Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, 500 AE Enschede, The Netherlands
\textsuperscript{2}Department of Computing Sciences and Mathematics, University of Stirling, UK

L.M.Buorianu@cs.utwente.nl, savi@cs.stir.ac.uk

Abstract. Nowadays formal software development is characterised by use of multitude formal specification languages. Test case generation from formal specifications depends in general on a specific language, and, moreover, there are competing methods for each language. There is a need for a generic approach to formal testing and for sound ways of combining test case generation methods. We address these issues using Correspondence Carrying Software, a paradigm for integrating heterogeneous specifications using viewpoints and category theory. We illustrate our approach with a small example of a train system specified using OCL and Z.

Keywords: formal testing, heterogeneous specification, category theory, relational algebra, Z, UML.

1. Introduction

Contemporary formal software development is characterised by a large heterogeneity, increasing its inherent complexity. Formal heterogeneity has been subject to many investigations and several main directions emerged. One is by integrating formal methods, and a series of conferences and workshops dedicated to this [5]. When more than two notations are involved, it seems unrealistic to integrate all of them. An alternative attempt is to create a semantic framework, in which all specification languages can be translated. There are several drawbacks of this approach: the frameworks are becoming extremely complicated and abstract because they aim to get very general (i.e. capability to translate many languages); during translation process, the initial structure of a specification can be lost; translation might be much more complex than the initial specification; in case an inconsistency between overlapping specification is detected, the generated feedback will expressed in terms of mathematics of the framework, instead of using initial languages. A natural candidate for a mathematical framework for heterogeneity results from combining category theory with logic (let us call categorical logic [29] such a hybrid). Indeed, logic is at the heart of any specification language and category theory was invented to express and analyse commonalities of different mathematical languages. Probably, the best known categorical logic is that of institution [23]. An institution uses the categorical language to model the concept of general logic introduced and studied by Ebbinghaus [20]. One of
the main contributions of institutions theory is the concept of translation between logics: a rich palette of categorical properties have been involved to describe the invariance of logical validity under translation. A major breakthrough into heterogeneity has been got when the concept of Grothendieck institution [11] was introduced by R. Diaconescu and further developed by T. Mossakowski [28]. It provides a consistent way of using multiple logics in software development conditioned by the possibility of moving across logics. Despite notable applications to structured specification and heterogeneous software components, this approach suffers drawbacks similar to those of integrated frameworks. Although the concept of institution is relatively simple (it makes use of concepts of category and functor only), the logic translation and Grothendieck construction [11] concepts give rise to categorical theories of remarkable complexity. Moreover, not all specification languages admit institutional semantics (this is the case of most of synchronous languages and process algebra), and not all institutions admit translations or translations with nice properties. Our experience of working with these approaches suggests that there is always a price to pay when dealing with heterogeneity: a more abstract concept specification language can simplify the translation counterpart of the theory. Moreover, an adequate methodology can drastically reduce complexity. The categorical logic we use to formalise spec languages is provided by categorical type theory and the methodology we use is based on viewpoints. Their combination results in the Correspondence Carrying Software paradigm (abbreviated CoCS) [10].

No prior knowledge of CoCS is necessary to understand this paper. The CoCS methodology is used rather semiformal; the full categorical account will subject of a further paper. A specification language is formalized categorically as a development frame (in Section 3), where development relations (like refinement) - a cornerstone of viewpoints methodology - play a primitive role. Development frames are classified in two main classes - constrained oriented (abbr. COD) and metamodeling oriented (abbr. MMD) - dual to each other. A major repercussion of this duality is on viewpoint unification: it is constructed by colimits (mainly pushout) in COD frames and by limits (product, pullback) in MMD frames.

Testing is a laborious work and it is the most time consuming activity of the practical verification. Testing might benefit considerably using viewpoints. Although viewpoints as oriented specification techniques are used in a great variety of development technologies, there is no mature use of them in testing. The earliest discussion on particular problems in testing viewpoint developed systems appears in [24]. To our knowledge, the first formal attempt on viewpoint testing was reported in [11], and further continued in [34]. The most difficult issue surrounding the application of viewpoints to testing is how test cases can be consistently and non-redundantly combined. This difficult problem is solved in our approach using category theory. It is really difficult to put together the abstractness of category theory with the constructivism required by software testing. The first attempt was reported in [3], followed by [25]. In [11] formal testing in a Grothendieck institution is developed. These approaches are mostly formalising the relationships between test cases and formal semantics. There are two main directions in formal testing: one is based on abstract data types (it corresponds in our approach to COD frames); and another one is based on labelled transition systems [31] (and it corresponds in our approach to MMD frames). In software practice formal methods having algebraic and coalgebraic nature are used not only all together but also integrated. Model-oriented
approaches as those based on languages like Z, VDM, B are neither algebraic or coalgebraic, but we can discover both aspects in their logical methods. In this situation category theory rediscovering what was constructively done using set theory: categorical relational algebra can express the logical content of both algebraic and coalgebraic methods. The implications of the mathematical exercise turn out to be constructive: it offers a way to integrate specification languages as described in [9] and a sound basis to translate languages based on algebraic specifications and process algebra into Z.

To formalize categorically the concept of test case, we use the development frames. The test cases are associated to syntax via a functor and therefore, we can assess categorically their correctness with respect to a standard (categorical) semantics.

The paper is structured as follows. In the next section we give a brief summary of a categorical logic approach to viewpoint specification. In Section 3 we present the formal testing theory. Section 4 presents an illustrating example: a small embedded system. The partial conclusions of this formal experiment are sketched in the last section.

2. A Categorical View on Specification Languages

2.1. Category Theory Concepts and Notations

Let SET be the category of sets and functions and REL the category of sets and relations. By Pr eOrd, we denote the category of preorders and sup-preserving functions. The powerset endofunctor is denoted ψ. The application of a functor F to an argument X is denoted by F[X]. A fibration from the total category E to the base category B is a functor Fib : E → B for which, for every e ∈ E | and u : t → Fib[e] ∈ B, there is a Cartesian lifting of u [1].

A posetal hyperdoctrine is a contravariant functor ψ : Bop → Pr eOrd, , where B is Cartesian (i.e. it has all finite products). We assume that, for each arrow f ∈ B, the monotone function ψ[f] preserves sups and has a left adjoint, written ⊥f (defining quantification as adjoints of projections is the essence of categorical logic). We also require the Beck-Chevalley and Frobenius conditions [29]. Hyperdoctrines form a special case of fibrations.

Binary relations are predicates whose type annotation corresponds to a product of contexts. If the base category has binary products, the (categorical) logic of relations can be obtained from (categorical) predicate logic considering only those predicates which are relations. Categorically, the category of binary relations REL_E and single carrier binary relations SREL_E arise from E; by pullback or change of base in a regular fibration Fib [29]:

\[
\begin{array}{c}
SREL_E \xrightarrow{\Delta} REL_E \xrightarrow{\rightarrow} E \\
\downarrow \quad \Delta \downarrow \quad \downarrow Fib \\
B \xrightarrow{A \rightarrow (AA)} B \times B \xrightarrow{\rightarrow} B
\end{array}
\]

2.2. Relational Data Types

In this section we present a categorical formalisation of specification languages and relational data types [18], using the categorical logic of relations [29]. The formalism, in
the set theoretic variant, was applied to UML in [6]. Semantics of many specification languages, as Z, OCL, and (partly) unified theory of programming can be formalised as relational abstract data types that can be thought of spec languages structuring categories.

For any relation \( \rho \in \text{REL}_E \) we define its precondition as \( \text{pre}_\rho \cap \text{id} \cap \rho^{-1} \) and its image by \( \text{im}_\rho = \text{id} \cap \rho^{-1} \rho \). Obviously, \( \text{pre}_\rho \) and \( \text{im}_\rho \) can be seen as subobjects in \( E \) (for example \( \text{pre}_\rho \cap \text{id} \) and \( \rho^{-1} \rho \) can be identified with the set \( \{ c \mid \{ c \} \in E \land (c, c) \in \text{id} \cap \rho^{-1} \rho \} \)). A postcondition \( \text{post}_\rho \) of \( \rho \) is a relation (predicate) between \( \text{pre}_\rho \) and \( \text{im}_\rho \).

Let \( G \) be an object of global states and \( \text{State} \) be an object of local states in the discrete \( 2 \) objects category. A relational data type is a quadruple \( (\text{State}, \text{Init}, \{ \text{Op}_1 \}_{i \in I}, \text{Fin}) \), where \( \text{Init} \) is a binary relation from \( G \) to \( \text{State} \), every \( \text{Op}_i \) is a total single carrier binary relation on \( \text{State} \), and \( \text{Fin} \) is a binary relation from \( \text{State} \) to \( G \). A relational data type is canonical if all \( \text{Init}, \{ \text{Op}_i \}_{i \in I}, \text{Fin} \) are maps. Two relational data types are conformal if they have the same global state space and they are indexed in the same way.

A program \( C_p \) is a finite sequence \( p \) of operations (total single carrier binary relations from \( \{ \text{Op}_1 \}_{i \in I} \)), interpreted as their sequential composition. A complete program is a composition of relations \( \text{Init} \cdot P \cdot \text{Fin} \), where \( P \) is a program. An abstract data type (ADT for short) is a (possibly infinite) set of programs each one with initialization. A relational data type \( C \) data refines a relational data type \( A \), and we note this by \( A \sqsubseteq C \). If, for every finite sequence \( p \) over \( I \), \( C_p \sqsubseteq A_p \). For discussion about ADTs and concurrency, we refer to [18] and [7].

### 2.3. Categorical viewpoint specification

A spec space \( (\mathcal{S}, \sqsubseteq) \) is a complete join semilattice. Elements of \( \mathcal{S} \) are specifications and \( \sqsubseteq \) models the refinement relation. A spec space morphism is a map \( m : S_1 \to S_2 \) such that \( m(\sqcup F) = \sqcup m(F) \) for each \( F \in \mathcal{F} \). Spec spaces and their morphisms form a category \( \mathcal{S} \).

A constraint oriented development frame (COD frame, for short) \( (\mathcal{SS}, \mathcal{B}, F, \varphi, \text{Sem}) \) consists of

- A subcategory \( \mathcal{SS} \) of \( \mathcal{S} \)
- A category of vocabularies (signatures) \( \mathcal{B} \)
- A fibration \( F : \mathcal{SS} \to \mathcal{B} \)
- A posetal hyperfibration \( \varphi : \mathcal{B}^{op} \to \text{Pr eOrd}_\varphi \)
- A functor \( \text{Sem} : \mathcal{aS} \to \text{Pr eOrd}_\varphi \), such that \( \text{Sem}[Sp] \subset \varphi[F[Sp]] \).

The intuition behind the definition of COD frames is that the objects of \( \mathcal{aS} \) are axiomatic specifications, closed under consequence relation. The fibration \( F \) describes the vocabulary used to build a specification, and the hyperfibration \( \varphi \) describes all possible structures implementing a given vocabulary. From these structures, the functor \( \text{Sem} \) selects those that are precisely the models of a specification.

The most obvious example of posetal hyperfibration is the powerset functor, essential in modelling the Z's type system in [10]. Many important examples from type theory can be found in [15].

Most of algebraic specification and model oriented languages are instances of COD frames. In fact, there is an instance of COD frames, called specification frame [9] that considers also a local type structure and makes possible the interpretation of types.
as specifications as in Z [10]. In [9] it is shown that institutions [23] are instances of specification frames.

The contravariance of the hyperfibration $\varphi$ means that in the refinement process one gets less models by adding more formulas to specification. Adding one more formula to a specification means a new constraint on the system to be developed.

OCL constitutes an important motivation for generalizing institutions to spec frames. It has a rich collection of types, and some of them are very complex, as model types. Semantics of OCL types can be very diverse, including both algebraic semantics in the tradition of institutions (integers, reals, sets, bags) and behavioral semantics as various extensions of labelled transition systems (statecharts, MSCs, etc.). Therefore, a rigorous algebraic semantics for OCL must find the right level of abstraction, and spec frames were designed exactly for this purpose. A COD frame for OCL without model types is easy to construct based on the standard semantics. OCL model types require a specification frame for UML. A fully worked out frame would imply to largely repeat documents like [14]. We leave as an exercise to check that OCL formal semantics form a frame. The full categorical formalization of OCL as a COD frame will be given in a forthcoming paper. The first steps were started in [6, 9, 10].

A meta-model development frame (MMD) frame for short \{SS, B, F, $\varphi$, Sem, MOD\} has almost the same definition as constrained oriented development frame, except the hyperdoctrine component, which is nonposetal, pseudofunctorial and defined on the dual category of vocabularies: $\varphi : B \rightarrow \text{MOD}$, where MOD is some subcategory of CAT, usually the category of Cartesian closed categories.

Important examples of MMD frames are the (higher order) categorical logics of labelled transition systems (LTS) and their many variants. An LTS $P$ is a diagram $\Sigma _{P} \rightarrow \lambda T_{P} \Rightarrow \delta S_{P} \leftarrow \iota$ in $B$. The operations $\lambda$, $\delta$, $\iota$ and $\rho$ assign to each transition, respectively, a label, a source and a target. The constant $\iota$ is the initial state. A morphism $\varphi : P \rightarrow Q$ of labelled transition systems is a triple of functions $\Sigma _{P} \varphi : \Sigma_{P}$, $\delta _{P} \varphi : \delta_{P}$, $\iota _{P} \varphi : \iota_{P}$, $\rho _{P} \varphi : \rho_{P}$). The latter two form a graph morphism, preserving the initial state and the labelling - in the sense that an $a$-labelled transition is mapped to a $\Sigma _{P} \varphi (a)$-labelled one. With these kind of morphisms the labelled transition systems form a category LTS. It is fibred by the functor $\Sigma _{\text{LTS}} : \text{LTS} \rightarrow B$, which projects each labelled transition system to the corresponding alphabet. $\Sigma _{\text{LTS}}$ is a regular fibration [29]. There is a regular fibration $\Sigma _{\text{LTS}} : \text{LTS} \rightarrow B$, spanned by reachable labelled transition systems, where each state can be reached from the initial state. The inclusion $\Sigma _{\text{LTS}} : \text{LTS} \rightarrow \text{LTS}$ has a right adjoint, i.e. reachable labelled transition systems span a coreflexive subcategory of labelled transition systems.

The covariance of the hyperfibration $\varphi$ means that in the refinement process one gets less models by eliminating formulas from specification. Adding one more formula to a specification means a new building rule for the models of the system to be developed. In an MM1 frame the models of specifications are not specified directly, but rules to build them are described instead. This style is specific to meta-modeling. The most prominent examples of this style are the UML grammars, graph (algebra) transformation systems and the structured operational semantics [10].
Viewpoints are a fundamental vehicle in UML. Every diagram expresses a particular view on the system, and viewpoint integration is usually done by formalizing different UML diagrams in a single, monolithic formalism, like Object Z. UML was extended with model viewpoints in [6], and package semantics was formalized in a generic manner using relational data types. In [9], UML viewpoint integration is made more general, managing heterogeneity using category theory. In contrast to [6], where UML viewpoints have a similar structure, in [9] a hierarchical integration mechanism is defined.

From now on we consider a fixed COD frame $\mathfrak{G} = (\mathfrak{A}, \mathfrak{B}, \mathfrak{F}, \varphi, \text{Sem})$. Let $V \in \mathfrak{B}$.

A $V$-specification in a development frame (or $\mathfrak{G}$-viewpoint) consists of the vocabulary $V$ and a set of formulas formed out of $V$, closed under logical consequence. A $\mathfrak{G}$-viewpoint $\mathfrak{R}$ refines (by model containment MC) a $\mathfrak{G}$-viewpoint $\mathfrak{V}$ if they have the same vocabulary and $\mathfrak{F} \models \mathfrak{R} \subseteq \mathfrak{F} \models \mathfrak{V}$ and $\text{Sem}[\mathfrak{R}] \subseteq \text{Sem}[\mathfrak{V}]$.

We adopt the viewpoint consistency expressed in [8] by “A collection of viewpoints is consistent if and only if it is possible for at least one example of an implementation to exist that can conform to all viewpoints.”

The correspondences between viewpoints are expressed using diagrams, specifically spans (more precisely relations) from the correspondence specification to the viewpoints.

The unification of two viewpoints is defined as the smallest common refinement [8]. The unification of viewpoint based on MC refinement is constructed by taking the pushout of the correspondence diagram. This construction is more suitable for algebraic types [11].

For abstract data types, the MC refinement is not strong enough to provide unification. A suitable notion of refinement is the relational refinement [18]. There are many examples of relational data types that are data refinement but not MC refinement. This is also because MC refinement cannot be defined mostly for COD frames. Many refinement relations, for example from process algebra, have particular interpretation of the concept of “model” [7]. In this case, the unification is constructed following [8, 6]. The unification $\mathfrak{S} \cup \mathfrak{R}$ of two state spaces $\mathfrak{S} \cup \mathfrak{R}$ connected by a relation $\rho \subseteq \mathfrak{S} \times \mathfrak{R}$ is given by

$$
\mathfrak{S} \cup \mathfrak{R} = \rho \cup (\text{dom}^{-1}(\rho) \times \{\bot\} \cup (\text{ran}^{-1}(\rho) \times \{\bot\}$$

The unification $\mathfrak{U}_{A,B}$ of transitions $A$ (acting on $\mathfrak{S} \cup \mathfrak{R}$) and $B$ (acting on $\mathfrak{S} \cup \mathfrak{R}$) is

$$
\text{pre}_{\mathfrak{U}_{A,B}} = \text{pre}_A \lor \text{pre}_B,
\text{post}_{\mathfrak{U}_{A,B}} = (\text{pre}_A \Rightarrow \text{post}_A) \land (\text{pre}_B \Rightarrow \text{post}_B),
$$

where $\text{pre}_A$ (resp. $\text{post}_A$) is the precondition (resp. postcondition) of $A$ etc.

There is also an orthogonal (or vertical) dimension of unification. A viewpoint (describing, for example, complex dynamics) can be constructed on top of another viewpoint (describing, for example, complex data). In [9] a relational integration mechanism is described, and the specification language $Z$ is interpreted from this perspective.
3. Formal Viewpoint Testing

Formal testing in a COD frame can be easily inherited from the testing theory of algebraic specification [3], [26]. Homogeneous viewpoint testing have been developed in [9]. Viewpoint testing in MMD frame can not be easily inherited from the theory of its testing [31]. There is very little categorical foundation (not known to the authors). Moreover, unification in an MMD frame is a pullback and it was very recently understood. Fortunately, in [21] a uniform view on testing data and processes is proposed and this theory constitutes a good foundation for heterogeneous viewpoint testing.

3.1. Testing Specifications in a Frame

Let us consider an arbitrary MMD frame. We develop our formal approach to software testing in the algebraic tradition initiated by M.C Gaudel, et. al. in [3,22].

A testing hypothesis describes assumptions about the system or the test process to reduce the size of the test set. We consider here only uniformity hypotheses: we define subdomains of interpretation for items, such that the system has the same behavior for all the values of the sub-domain. Then we assume that testing for a value of a sub-domain is enough to test for all its values. The sub-domains are defined in a logical form by the concept of test cases.

A test instance is a couple \((C, D)\) where \(C\) is a test case corresponding to a test hypothesis and \(D\) is a test data defined with respect to this test hypothesis. Whenever \(F[C] = V\) we call \(C\) a \(V\)-test case (and analogously, for \(D\)).

The category of \(V\)-test cases, denoted \(\mathbf{TC}_V\), has: (1) objects: test cases; (2) morphisms: for a test case morphism \(C \rightarrow C'\), at each instance of \(C\) we associate an instance of \(C'\) with the same item, i.e. an inclusion between the set of instances of both test cases. \(\mathbf{TC}_V\) is a finitely cocomplete category: initial object is an empty test case, pushout is defined by putting together the instances of both test cases without repetition of shared instances.

The definition of \(\mathbf{TC}_V\) is very natural in context of COD frames, as developed in [111]. The theory of testing in MMD frames is obtained by categorical duality. But what means a test case in this case? An answer is offered in [21] and it consists in finite traces. The full categorical study of finite traces is presented in [33].

We define a Testing functor giving for every vocabulary its category of \(V\)-tests

\[
\begin{align*}
\text{Test} & : \mathcal{B} \rightarrow \text{CAT} \\
\text{Test}(V) & = \mathbf{TC}_V
\end{align*}
\]

such that the following satisfaction condition for tests is satisfied

\[
M' \simeq_V \text{Test}(\sigma)(a) \Leftrightarrow \text{Sem}(\sigma)(M') \simeq_V a.
\]

In a categorical specification logic, a program is identified with a viewpoint \(Psp\) and its execution with the semantics of the specification \(\text{Sem}[Psp]\).

Thus, a test goal is a formula \(a \in Psp\), and the satisfaction relations \(\text{Prog} \simeq_V D\) and \(\text{Prog} \simeq_V C\) are replaced by the test satisfaction relation for models \(\simeq_V \subseteq \varphi[V] \times | Test[V]|\) for each \(V \in \mathcal{B}\).
3.2. Testing Unification by Unification of Tests

Tests are derived from viewpoint specifications according to the formal language used by each team. Viewpoints could have different concepts (resp. describing languages or deriving techniques) for tests. A large heterogeneity of viewpoints, and implicitly of tests derived from them, implies the necessity of a common definition and understanding of them, as well a set of tests for the whole system description (i.e. unification).

The unification of viewpoints in MMD frame is a limit [10], the categorical analogue of conjunction. This categorical operation applies also to viewpoint test cases unification.

Suppose we have generated the test instances \( (C_{Vp}, D_{Vp}) \) and \( (C_{Vp'}, D_{Vp'}) \) from the viewpoints \( Vp \) and \( Vp' \), correct for the test goals \( X_{Vp} \) and \( X_{Vp'} \) and similar for the specification \( Co \), which expresses the correspondences between the viewpoints. It is important to note that we do not use any hypotheses about the way tests have been obtained. For example, \( D_{Vp} \) could be obtained from a correctness proof, as in [27], and \( D_{Vp'} \) could be obtained by any method of generating test cases from a formal specification.

Consider the unification obtained by the following pullback of viewpoints

\[
\begin{array}{ccc}
C_{\text{Unif}} & m_1 & \rightarrow & m_2 \\
\downarrow & & & \downarrow \\
Vp & & \leftarrow & Vp' \\
\downarrow m_1' & & \nearrow m_2' \\
\text{Unif} & & & \\
\end{array}
\]

(1)

corresponding to the pushout of signatures

\[
\begin{array}{cccc}
F[C_{\text{Unif}}] & F[Vp] & \sigma_1 & \sigma_2 \\
\downarrow & \sigma_1 & \downarrow & \sigma_2' \\
F\text{Unif} & F[Vp'] & & \\
\end{array}
\]

(2)

We are interested in the way \( (C_{\text{Unif}}, D_{\text{Unif}}) \) for \( \text{Unif} \) can be obtained. The correctness of the method results from the following proposition. This is a dual instance of more general results that have been proved in [11].

**Proposition 1** The test instance \( (C_{\text{Unif}}, D_{\text{Unif}}) \), obtained by pullback

\[
\begin{array}{ccc}
(C_{C_{\text{Unif}}}, D_{C_{\text{Unif}}}) & t_1 & \rightarrow & t_2 \\
\downarrow & & & \downarrow \\
(C_{Vp}, D_{Vp}) & & \leftarrow & (C_{Vp'}, D_{Vp'}) \\
\downarrow t_1' & & \nearrow t_2' \\
(C_{\text{Unif}}, D_{\text{Unif}}) & & & \\
\end{array}
\]

is correct for \( F^{-1}[m_1'](X_{Vp}) \land F^{-1}[m_2'](X_{Vp'}) \).

The proof is constructed following the pullback construction 1. Unlike pushout in COD frames, pullback in MMD frames is more complicated, being derived from **SET**. A
pullback construction for elementry (Petri) nets is described in [2]. Although the construction can be generalised to lts, we do not follow this way because it is very complicated. We use instead the category of $\varphi$–coalgebras.

The category theory of coalgebras is very rich, and no attempt of defining a theory of testing was made. The advantages of this approach are that syntax is considered (missing in the coalgebra approach), and the theory is smoothly compatible with relational algebra. We recall the correspondence between coalgebras of $\varphi$ and labelled transition systems $\langle S, \rightarrow, L \rangle$, $\rightarrow \subseteq S \times I \times S$. Define $B[L] = \varphi L \times X \rightarrow \text{REL}_{[L,X]}$ for any set $X$. The $B$–coalgebra associated to lts is $\langle S, \alpha_S \rangle$ with $\alpha_S : S \rightarrow B[S]$, $\alpha_S(s) = \{ (a, s') \mid s \xrightarrow{a} s' \}$. To any $B$–coalgebra $\langle S, \alpha_S \rangle$ we can attach an lts $(S, \rightarrow, A)$, by defining $s \xrightarrow{a} s' \iff \langle a, s' \rangle \in \alpha_S(s)$. In [30], p. 29, it is shown that the pullback of $F$–coalgebras can be obtained by applying directly the functor $F$ to all components of a pullback diagram in $\text{SET}$, conditioned by $F$ preserves pullbacks. For $\varphi$ there is given a simple characterisation at p. 30. The unified lts is obtained from the coalgebra resulted by pullback.

In [9], it is shown that abstract data types are specified by an hierarchical integration of an MMD frame on top of a COD frame. In this case, the test case integration becomes more complicated, and requires a DNF type method, described below.

3.3. A Generic DNF Method for Relational Data Types

The DNF method was defined in [16]. In the sequel, we define a generic DNF method, for generating test cases from a categorical relational type. The method produces a finite state machine (FSM) by analysing states and operations together. The category of test cases $\mathcal{T}_C$ we consider is the category of finite state machines, as defined, for example, in [1], p. 61. This category is in fact a full subcategory of $\text{LTS}^+$, where objects are ”specialised” by interpreting the transition relation as a monoid of actions. This generic method can be instantiated for specifications written in OCL, Z and VDM.

The main activities of the method comprise:

1. **Partition analysis of all operations.** This involves reducing the mathematical expressions defining an operation into a Disjunctive Normal Form (DNF) which gives disjoint partitions representing domains of the operation that should be tested. This step gives the transition in the FSM.

2. **Partition analysis of the system state:** The mathematical expression defining the system state is reduced into DNF, which yields disjoint partitions of the state values. This step gives the states in the FSM.

3. **Scheduling of tests** of different operations to avoid redundancy in the testing process. This step constructs the FMM by resolving transitions against states.

In a further step, test data are generated for use in the validation of the implementation.

4. Example

We illustrate our proposed methodology by a (simplified) case study of an automatic rain train driver. For more expanded case studies in this field, we refer to [17]. This is a
4.1. Basic Operation Viewpoint

This viewpoint describes the main functionality of a controller for a train stopping in a station.

There are three states, namely: W_o_d stands for ‘Wait open doors’, C_d_s stands for ‘Control distance and speed’, C_i stands for ‘Control immedicen’. Doors may not be opened on the way side, except when the train is stopped on a side-track. Platform doors should not normally be opened while the train is moving.

\[
\begin{align*}
\text{Train} & : \text{OCLTransition()} \\
\text{pre state} & = \text{W_o_d and located and in_station} \\
\text{post state} & = \text{C_d_s}
\end{align*}
\]

where located means that the locating system works properly and in_station means that all platform-side doors face the platform.
A graphical viewpoint description is depicted in Fig. 2 (that contains more transitions, resulted by considering more viewpoints).

Applying the test generation method [27] one gets the following test data for operation $ac$

$T_1 = (loc : Boolean = true, loc : Boolean = true, st : state: st@pre = a: st = c)$

### 4.2. The Safety Viewpoint

The stakeholders of this viewpoint focus on the safety aspects only. In order to increase the passenger flow, the doors may be opened in station when the speed is less than some value $spd_{-}open_{-}doors$, thus anticipating the halting. But care must be taken to ensure safety:

- the train must not accelerate again beyond the speed limit above;
- it must stop before it has covered some distance limit;
- it must not pass the end of the station with its doors opened. When a forbidden situation happens, the alarm of all doors is set and emergency braking occurs.

The viewpoint adds a new state, $E_{-}S$, for ‘Emergency stop’, more transitions and more firing conditions to those of basic operation viewpoint.

$States = \{E_{-}S, C_{-}i, C_{-}d_{-}S, W_{-}a_{-}d\}$
In Fig. 3, a graphical description of the viewpoint is given.

As in Z, OCL sets are typed (even finite sets can be considered as types). This is useful to define a test case notation, and it can provide a conceptual and notational basis to construct a unification algorithm for test cases using pushout.

Applying the method from [13] one gets the following test data for operation ac

\[ \langle \text{loc} : \text{Boolean} \quad \text{true}, \text{loc} : \text{Boolean} \quad \text{true}, \text{st} : \text{state}, \text{st@pre a}, \text{st c}, \text{speed} : \text{AnalogReal} \quad 15, \text{cl_w} : \text{Boolean} = \text{false}, \text{cl_w} : \text{Boolean} = \text{false} \rangle \]

4.3. Viewpoints unification

We have done the unification simpler by making correspondences implicit, using the same name for items that should be put in correspondence. The unified vocabulary (signature) \( UTrain \) is obtained by the pushout (2) from the viewpoint vocabularies (i.e. it is the “minimal” vocabulary containing non-redundantly all viewpoint vocabulary components).

\[
\begin{align*}
\text{ZTrain} & \quad \text{ZInit} \\
\text{state} : \text{States} & \quad \text{state}' = W\_d\_d \\
\text{speed, spd_open_doors} : \text{Real} & \\
\text{located, closed_pf, closed_way} & \\
\text{in_station} : \text{Bool} &
\end{align*}
\]

\[
\begin{align*}
\text{ZTransition} & \\
\text{\Delta ZTrain} & \\
\langle \text{state} - (C\_d\_s \land \text{speed} = 1) \land \text{state}' = C\_i \rangle & \\
\lor & \\
\langle \text{state} - C\_i \land \text{closed_pf} \land \text{closed_way} \land \text{state}' = W\_d\_d \rangle & \\
\lor & \\
\langle \text{state} = W\_d\_d \land \text{located} \land \text{in_station} \land \text{speed} < \text{spd_open_doors} \land \text{closed_way} \land \text{closed_pf} \land \text{state}' = C\_d\_s \rangle & \\
\lor & \\
\langle \text{state} = W\_d\_d \land \lnot \text{closed_pf} \land \text{closed_way} \land \lnot \text{located} \land \text{in_station} \land \text{closed_way} \land \text{state}' = F\_s \rangle & \\
\lor & \\
\langle \text{state} = W\_d\_d \land \lnot \text{closed_pf} \land \text{closed_way} \land \text{speed} = 1) \land \text{state}' = C\_i \rangle &
\end{align*}
\]

The common operations have the same postcondition, simplifying the unification formulas. The unified statechart, resulting from applying the method from [6], also described in Section 2.C, is given below:
Figure 4: Viewpoint Unification

\[\text{UTransition = UTransition} \lor [\Delta UTrain | \text{speed} = 0] \lor \\
\left( (\neg \text{in\_station} \lor \text{located}, \land \text{closed\_way} \land \text{closed\_pf}) \right)\]

where UZTransition is ZTransition with signature extended to \(\Delta UTrain\). A graphical illustration of the unification appears in Figure 4.

The test cases for the unified operations can be obtained by pullback from viewpoint test cases for corresponding operations. In our example, the test case for operation \(ac\) is the pullback of \(T1\) and \(T2\). Because of blocking interpretation, this pullback must be \(\{\text{and actually it is}\} T2\).

There is a subtle aspect concerning the precondition interpretation. UMI describes the failure of precondition as a failure of that operation. Most authors agree that this circumstance must be managed by the implementors. In the context of viewpoint development, two interpretations are possible [18]:

- underspecification: the viewpoint stakeholder does not have enough information about precondition, and it is assumed that the completing information is given in a different viewpoint. This is the case of first viewpoint, where basic functionality is given and thus specific information should be added by other viewpoints.

- blocking: an operation must be blocked outside precondition. This is the case of second viewpoint, where safety conditions demand the system is unsafe outside preconditions.

The presence of blocking interpretation in viewpoints extends to the unification and imposes extra consistency conditions. In the example we considered, unification exists if for every common operation, the preconditions in the first viewpoint are included into those of the second, and this is indeed the case. Constructive consistency check concerns consistency issues like completeness and non-contradiction of (viewpoint) preconditions. These can constitute suitable test purpose criteria. We also have to consider a uniformity hypothesis on the domain of each precondition, carefully chosen, as too strong uniformity hypothesis can weaken the generated test cases.

A further advantage of constructing test cases for unification by unifying test cases derived from viewpoints is that we can use different test derivation methods.
5. Conclusions

This work is a natural continuation of our papers [10, 9, 11, 6, 27]. The main issue we address is the sound testing methods combination (and their associated tools) using viewpoints. Many automatic procedures for generating tests from formal specification has been developed, and combining their outputs as result of a distributed approach to formal specification is a necessary step. Our approach is made generic using category theory.

A major motivation for CoCS is UML. There is an explosive development of UML formalisation and test case generation methods. In fact, any formal specification notation was used to formalise UML views and as a basis for test case derivation. This work does not propose any of these, but rather our goal is to find an ‘ecumenical’ way to live with this extreme formal heterogeneity. A mathematical tool that could deal with multiple formal languages and methods should have a high level of abstractness. For many years category theory has been proposed as mathematical framework that treats uniformly different aspects of formal notations. We adhere to this trend, because the categorical language can offer deep insights on relatively different theories. There are now many papers and monographs introducing the basics and the subtleties of category theory to computer scientists and software engineers. In the limited space of this paper it is difficult to provide a full list of references, but the interested reader may find them quite easily.

The mathematical properties are not so easy to be expressed and a difficult choice is concerning how deep category theory should be used. Other categorical approaches to formal heterogeneity like those based on Grothendieck institutions [28] suggest that there is a level of complexity, which makes a light categorical use almost impossible. Consequently we keep the use of categorical algebra at the necessary level of abstraction. As in pure mathematics the categorical methods will eventually end in more intuitive set theoretic models. This final step is absolutely necessary in order to make them used. But, this final step will be done in a following paper.

Further work:

One could observe that the viewpoints ADTs in the example are not conformal. They can be made conformal by considering stuttering steps, which in *are skip* $\exists$State]. The data refinement need to be generalised to the weak refinement [18]. Of course, the example could be simplified to have only conformal ADTs, but this is not the point. We consider it is more natural to start building a theory from simpler cases, next step of our research is to consider weak refinement.

An interesting issue appears when we consider examples where all viewpoints have underspecification interpretation. In this case, corresponding operations can have different preconditions. The full generality of pushout construction is necessary, and an algorithm to construct it (using its decomposition in disjoint union and equalizer) is useful. We have encountered a case where we have to work in the dual category of specification and downward simulations (a result from [18] ensures this category does exist). The precondition of the unified operation is then the conjunction of viewpoint preconditions, and therefore we have to replace disjoint union by product in the method of unifying test cases. We intend to continue this paper by more specific work in test case derivation for hybrid systems [12] and HybridUML [4] specifications. We have also developed a development frame dedicated entirely to hybrid systems, using modal logics. This can be
used in conjunction with HybridOCL, rising the issue of heterogeneous viewpoints.

References


[34] L. Wildman et. al.: Viewpoint-based Testing of Concurrent Components. In [5].