Extending UML Templates towards Computability

José Farinha[1] and Pedro Ramos[2]
[1]ISTAR, ISCTE-IUL, Av. Forças Armadas, Lisbon, Portugal
[2]IT-IUL, ISCTE-IUL, Av. Forças Armadas, Lisbon, Portugal

Keywords: UML, Templates, Verification, Computability.

Abstract: UML templates allow the specification of generic model elements that can be reproduced in domain models by means of the Bind relationship. Binding to a template encompasses the substitution of that template's parameters by compatible domain elements. The requirement of compatibility, however, is checked over by UML in a very permissive way. As a consequence, binding to a template can result in badly-formed models and non-computable expressions. Such option in the design of UML was certainly intentional and meant to allow for richer semantics for the Bind relationship, as the specialization of the concept is advised at several points of the standard. This paper proposes one such specialization. One that guarantees well-formedness and computability for elements bound to a template. This is achieved by introducing the concept of Functional Conformance, which is imposed between every template's parameter and its application domain substitute. Functional conformance is defined in terms of well-formedness rules, expressed as OCL constraints on top of OMG's UML metamodel.

1 INTRODUCTION

Through the concept of Template, UML allows the definition of generic solutions to recurring problems. An UML template is a model element embodying a patterned solution that can be reproduced within any domain model where the addressed problem is observed. This is achieved by binding a model element of that domain to the template, through a Bind relationship. In order to have a template reproduction contextualized to the target (domain) model, a template is a parameterised element. A template parameter marks an element participating in the template's specification that, in a reproduction of that template, must be substituted by an actual element in the target model. Only when all of the template's parameters are substituted, it becomes an actual, fully integrated solution in the target model.

Aiming at getting consistent specifications out of a template reproduction, UML enforces a set of constraints to template parameter substitutions. One such constraint imposes that a substitute element must be of the same metaclass as the parametered element: an attribute must be replaced by an attribute, an operation by an operation, etc. Another constraint ensures that if a parameter exposes a typed element, its substitute must have a type that conforms to that parametered element's type.

Yet, the set of validations falls short in guaranteeing the well-formedness of the element resulting from the template. For instance, UML allows an operation *Op1* be substituted by an operation *Op2* whose signature is not compatible with the former's. If *Op1* is substituted by *Op2*, every call to *Op1* in the template's code will be reproduced as a call to *Op2*…but with a set of arguments aligned to *Op1*'s signature, which makes such call to *Op2* badly-formed. Furthermore, UML allows the specification of a set of substitutions that are not mutually consistent. For instance, having a class and one of its attributes exposed as parameters in a template, UML allows the former being replaced by a class *C'* and the latter by an attribute that is not a member of *C'*. Merely considering the semantics and constraints that UML declares for the concept of template, according to version 2.1.4 of the standard (OMG 2012), it seems the language greatly relies on the modeller's skills and prudence.

In spite of the permissive set of validations, a bad substitution will generally be prevented by some well-formedness rule, associated to some element within the bound element that will try to use the bad substitute. In the example above, the substitution of *Op1* by *Op2* will generate errors raising from the

DOI: 10.5220/0005257101220133
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 122-133
ISBN: 978-989-758-083-3

calls to *Op2*, which will actually prevent the substitution. However, the error that will be reported will be detuned from the real source of the problem. The problem will be reported somewhat like "Arguments to *Op2* do not match that operation's signature". But the cause of the problem is the substitution of O*p*1 by *Op2*. Hence, although UML is not really trusting in the prudence of the developer, it certainly trusts in his/her ability to diagnose.

This paper proposes an additional set of constraints for the concept of Template, aiming at removing the aforementioned disadvantages. The set of constraints was designed with the following purposes: (1) Guaranteeing the well-formedness and computability of any element resulting from the application of a template; (2) Reporting problems resulting from incorrect usages of a template to their real causes, i.e., to inadequate bindings and/or substitutions.

In (1), 'well-formedness' means that none of the components of an element resulting from the template will violate any constraint imposed by the UML metamodel. 'Computability' means that every expression within the template or within the bound element can be processed and evaluated to a value (including *Null*), i.e., the expression successfully compiles in the scope of the model it belongs to. For simplicity, in this text, the term 'computability' will be used meaning 'well-formedness' as well.

To accomplish (2), the proposed constraints establish conformance criteria between every parametered element and its substitute. The way a parametered element is used by the template does not participate in the criteria. In that way, any error may be reported exclusively in terms of the adequacy of a substitute to a parameter, in the context of a specific binding to the template.

The constraints put forth in this text formulate a concept named *Functional Conformance*, a term aiming to denote the equivalence between two elements, from a third-party, client perspective. An element (the substitute) conforms functionally to another element (the parameter) if its characteristics and scope allow it being used instead of the latter. Functional Conformance is presented through its definition and several illustrating examples, which should provide an intuitive perception of its effectiveness as a guarantee of computability. A formal demonstration of that effectiveness is postponed to a future paper, due to lack of space.

The structure of the paper is as follows:

- Section 2 provides a brief introduction to the concept of Template in UML;
- Section 3 points out some problems in assuring that elements resulting from templates are well-formed and computable;
- Section 4 proposes the concept of Functional Conformance to ensure computability;
- Section 5 presents related work;
- Section 6 draws some conclusions on an empirical evaluation of functional conformance and outlines some prospective benefits of the concept;
- The appendix includes a set of OCL constraints that assess functional conformance.

2 AN INTRODUCTION TO UML TEMPLATES

In UML, a template is a parameterised model element that can be replicated and have its replicas contextualised to the models they are put into. Model elements of several kinds may be qualified as templates. For instance, classes, packages and operations are allowed to be declared as templates and, therefore, be reproduced as concrete classes, packages and operations, respectively.

The complete set of model element types that can be declared as templates – called templateable elements – is comprehended by the following metaclasses: *Classifier*, *Package*, *Operation* and *StringExpression*. *Classifier* encompasses all kinds of element that may have instances: Class, Datatype, Association, Use Case, Activity, etc. Package templates should be used when a model fragment encompassing two or more classifiers is meant to form a single template and be replicated as a whole. Operation templates lack of graphical notation but are supported by the UML metamodel. Finally, *StringExpression* templates are used to derive concrete element names or literal strings by concatenating the values of a template's parameters. *StringExpression* templates do not pose computability problems; therefore, they will not be referred from now on.

In this paper, the term "target" is used to refer to a model, package or class that gives context to a replica of a template.

Figure 1 and Figure 2 show a class and a package template, respectively. A template element is recognized graphically through the dashed rectangle on the top-right corner, whose purpose is to declare the template's parameters.

Template parameters declare some of the elements participating in the specification of the template as parametered. Such elements are said *exposed as parameters*. When applying a template, its parametered elements must be replaced by elements of the target model in order to obtain a fully functional and contextualized reproduction of the template. If any of the parametered elements is not replaced, the reproduction of the template is also a template – this allows the incremental definition of templates.

In this text, for clarity reasons, *parametered element* will sometimes be referred simply as *parameter*. However, it should be noted this is use of “parameter” in the broad sense, since there is a strict difference in UML between the parameter and the element it exposes: the parameter is a “mark” superimposed to the element, which qualifies it as replaceable when applying the template, and it can supply a default in case such element is not explicitly replaced.

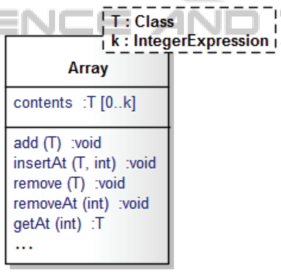


Figure 1: Example of class template.

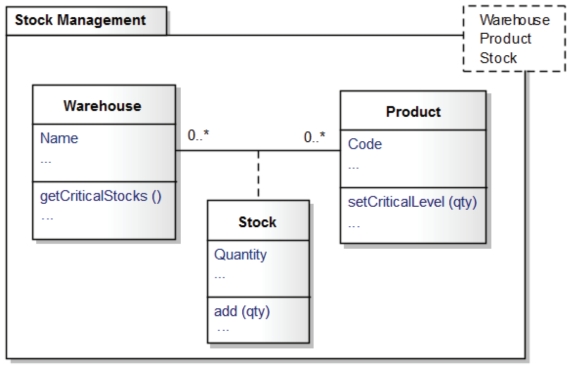


Figure 2: Example of package template.

The UML concept for assigning a target element to a template parameter is *Substitution*. It is said that the target element *substitutes* the parameter. The former is often called *actual parameter* and the latter *formal parameter*. In this text, for simplicity, an actual parameter will be referred as the *substitute*.

In Figure 1, class *Array* is a template with two parameters, *T* e *k*. *T*’s kind is *Class*, meaning it must be substituted by a class. *k*’s kind is *IntegerExpression* and therefore must be substituted by such expressions, including one with a plain literal value. In Figure 2, the package *StockManagement* is a template with three parameters – *Warehouse*, *Product*, and *Stock* – all of them exposing classes. When applied to a target package, its parameters require picking three target classes as substitutes. These substituting classes will receive the specifications of their respective parametered classes, and any relationships between these.

Both the name and the kind of a parameter are determined by the element the parameter exposes. In the example in Figure 1, parameter *T* exposes a class named ‘T’ (not shown in the figure) and parameter *k* exposes an integer expression named ‘k’. Parameters adopt the names of the elements they expose.

Any model element accessible from a template may be exposed as a parameter. For instance, in Figure 3 *AlphabeticList* is a class-template with one parameter that exposes another class. The dashed line labelled “exposes” is merely illustrative; there is no graphical notation in UML that links a parameter to the element it exposes.

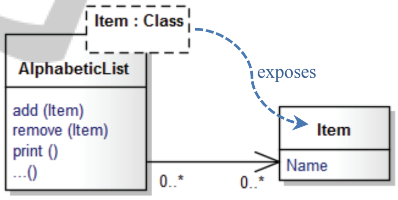


Figure 3: Definition of a template with a class-parameter.

A template may be used to specify elements from scratch or to add specifications to elements having specifications of their own. For instance, a class-template may be used to create a class as a replica of itself, as well as to add all its members (features, constraints, etc.) to an existing class. The application of a template is specified through a *Bind* relationship. A Binding is a directed relationship from a *bound element* to a template. By means of the Bind relationship, everything specified for the template is also valid for the bound element.

Figure 4 shows a binding to the *AlphabeticList* template, by class *AlphabeticList<Person>* (which is said anonymous). In that binding, the template parameter (*Item*) is substituted by class *Person*. The UML notation for a substitution is textual, in the form ‘parameter –> substitute’, placed next to the graphical representation of the binding. The figure also shows (on the right) the semantics of that binding: class *AlphabeticList<Person>* receives a reproduction of *AlphabeticList*'s specification (of all its features, behaviours, constraints, etc.) with all references to *Item* replaced by references to *Person*. It's worth noting that, among the features inherited by *AlphabeticList<Person>*, this class receives a copy of the association-end connected to *Item*; such copy will connect to *Person*, since this class substitutes *Item*. Strictly, the association-end is reproduced as a property and such property will not be part of any association. Nevertheless, the association symbol linking to *Person* is shown for a question of clarity.

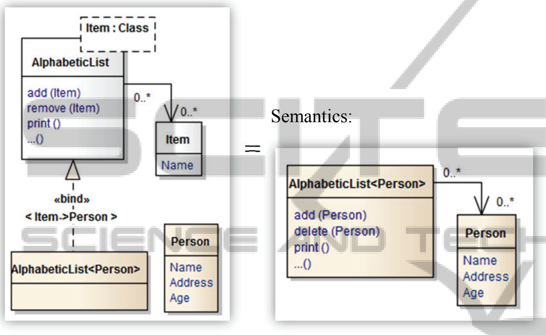


Figure 4: A bind and its semantics, producing an anonymous class.

Figure 5 shows another bind to the same template, with a bound class that is not anonymous, but named *Glossary*. Strictly according to the semantics of UML for the Bind relationship, the binding of *Glossary* to *AlphabeticList* is equivalent to the diagram on the right of that figure. In that diagram, *AlphabeticList<Concept>* is a non-referenceable auxiliary class, whose purpose is solely the formalisation of the semantics.

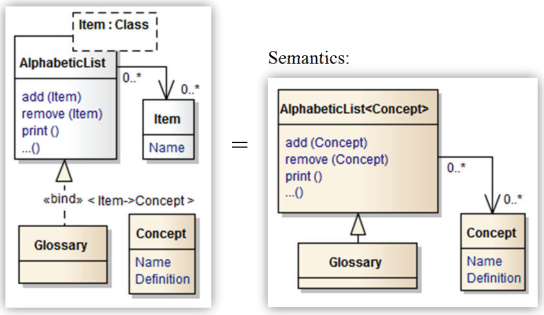


Figure 5: Another example of binding to a template.

Figure 6 shows a bound class with specifications of its own. In such cases, the bind merges the specification of the template with the contents of the bound class.

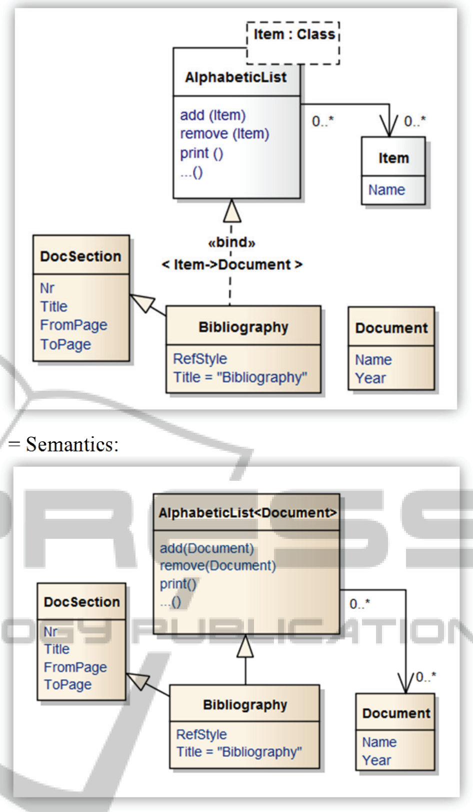
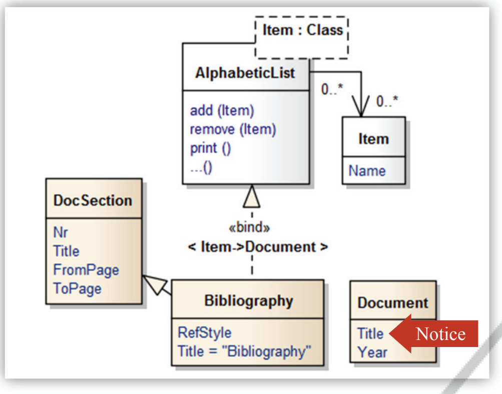


Figure 6: A bound class with contents of its own.

The purpose of the *AlphabeticList* template is to maintain a list of items sorted alphabetically. To perform such ordering, its operations use an attribute called *Name* in class *Item* (the use of *Name* is not observable in the figure). Since the code of *AlphabeticList* is copied to its bound classes, these will also use a *Name* attribute. Being *Item* substituted by another class, *Name* must also exist in this class. If it doesn't, the code of the bound class will not be compilable. (Notice that class *Item* is not part of the template; thus, *Item*'s contents will not be copied to the substituting class; this class must have a *Name* attribute of its own.) The situation is exemplified in Figure 7: since *Document* doesn't have a *Name* attribute (but one called 'Title', instead), every expression in the template referring to *Item::Name* will not be computable once reproduced in class *Bibliography*. For instance, expressions 'it.name' and 'iter.name' will not be computable in *Bibliography*.

Situations such as in Figure 7 require flexibility regarding the attribute corresponding to *Name*, in the substituting class. This is achieved exposing *Name* as another template parameter. The definition of the template becomes as in Figure 8.



= Semantics:

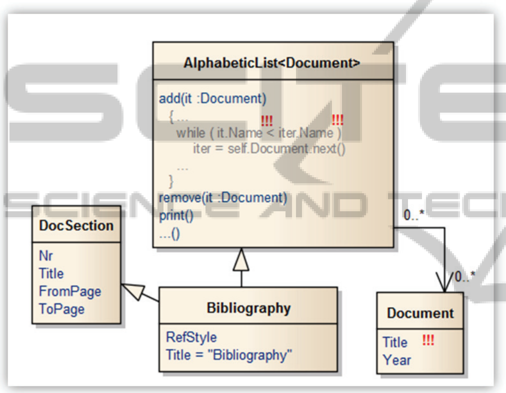


Figure 7: A bind that produces non-computable code.

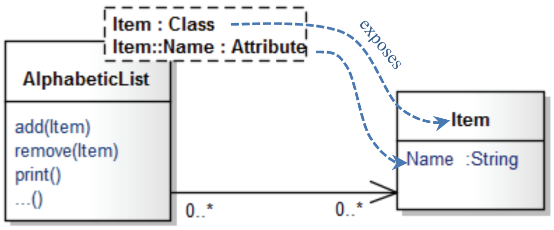


Figure 8: Definition of a template with a class parameter and an attribute parameter.Since the new parameter in Figure 8 exposes an attribute, it must be substituted by an attribute in the target model. For attribute-parameters, UML establishes also that the type of the substitute must conform to the type of the parametered attribute. Thus, *Name* must be substituted by an attribute whose type is *String* or a subtype of it.

Note: although the UML notation for a substitution is textual, in this paper, for clarity reasons, sometimes it is shown in a graphical way, draw as a dashed arrow (similar to a dependency) linking the parametered element to its substitute and labelled with “substitution”.

3 SOME LIMITATIONS OF UML TEMPLATES

Conformance of kind (class, attribute, package, etc.) and conformance of type (this for typed elements) are the only restrictions that apply to the substitution of UML template parameters. Consequently, not computable specifications such as the one previously shown in Figure 7 and the one in Figure 9 are considered valid bindings by UML.

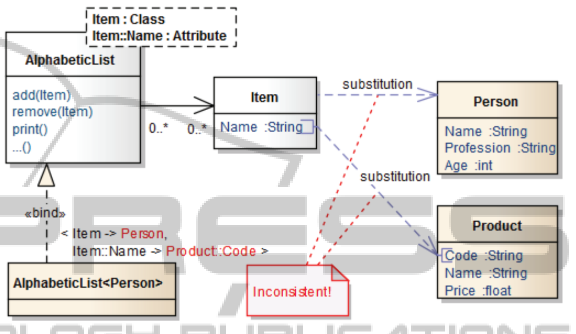


Figure 9: A bind that produces non-computable code.

Since the referred situations lead to badly-formed elements or non-computable expressions, some error will be reported. However, that error will not refer the binding, nor the substitutions it contains, as the source of the problem. Instead, it will refer problems within the bound element that are not immediately recognized as consequences of an erroneous binding or substitution.

For instance, the problem with Figure 7 is that template *AlphabeticList*, as defined in there, is not applicable to classes *Bibliography* and *Document*. An error message should report that and, more specifically, it could mention that the substitution of *Item* by *Document* cannot be done. Instead, the error that will be raised is about variables *it* and *iter* being unable to access an attribute called ‘Name’.

Similarly, the error raised in Figure 9 will mention that variables *it* and *iter* can’t access an attribute (*Code*), instead of the real cause: the incorrect substitution of the attribute *Name* of *Item* by an attribute not pertaining to the substitute of *Item* (*Person*). The rule that would signal correctly the problem would be: considering two parametered elements P_{child} and P_{parent} being both substituted in a binding, if P_{child} belongs to P_{parent} then the substitute of P_{child} must belong to the substitute of P_{parent}. Such rule should be a constraint to substitutions or to bindings, but no such constraint exists in UML.

Similar problems will arise if a single-valued property is substituted by a multivalued one, or if an operation is substituted by another with incompatible signature.

The examples given so far show that there are some reasonable constraints missing in UML templates. Although not strictly necessary to prevent badly-formed elements resulting from a template, their absence increases the risk of ill-specified binds and causes error-reporting dyslexia. Next section proposes a set of constraints that would remove these shortcomings of UML templates while ensuring the computability of bound elements.

4 FUNCTIONAL CONFORMANCE

The concept of *Functional Conformance* between two model elements is used to express that if one is used successfully by a template the other will also be used successfully in a reproduction of that template, if used in the same circumstances and with the same goals. Taking an example from a domain other than computer science, it can be said that a piano is functionally conformant to a clavichord, for the purpose of playing a classical piece of music. The piano may be used in today's reproductions of a Mozart piece, substituting the long ago used clavichord. Providing that it is used to play the keyboard line of the piece (it won't be functional to play the strings' part), it will produce the same results as the clavichord (or even better results). If the analogy is allowed with template-based software development, the parametered element in a template is the original "device" to which its substitute is expected to be conformant.

In the scope of a particular binding, an element of the target space functionally conforms to an element of the template space if it conforms to the former regarding type, multiplicity, contents, and staticity, and if it is visible from the bound element. The following subsections define these requirements for conformance.

Note: in some figures of this paper conformance is shown graphically as a dashed arrow, from the parametered element to its substitute, meaning that the latter conforms to the former. This graphical representation uses the reversed direction of that of the phrase "conforms to" for the sake of consistency with the direction of the UML notation for substitution (parametered –> substitute).

4.1 Type Conformance

Type conformance applies to every typed element: properties, expressions, constants, operation parameters, action pins, etc. This conformity criterion is partially enforced by UML, which states that the type of a substituting element must be the same or a subtype of that of the parametered element. However, the UML rule is incomplete, for two reasons: (1) UML only applies it to properties and value specifications (expressions and constants), (see constraints of *TemplateParameterSubstitution* and operation *isCompatibleTo()*, in (OMG 2012)); (2) this rule should be applied only if the type of the parametered element is not substituted.

Full type conformance should be: (1) imposed on all typed elements; (2) formulated considering two different scenarios:

- If parametered element e^{P} has a type that is not substituted: element $e^{\odot}$ conforms in type to e^{P} if its type is the same or a subtype of e^{P}'s. This is the original UML constraint.
- If e^{P}'s type is substituted: element $e^{\odot}$ conforms in type to e^{P} if its type is the same or a subtype of the substitute of e^{P}'s type.

The second scenario is exemplified in Figure 10.

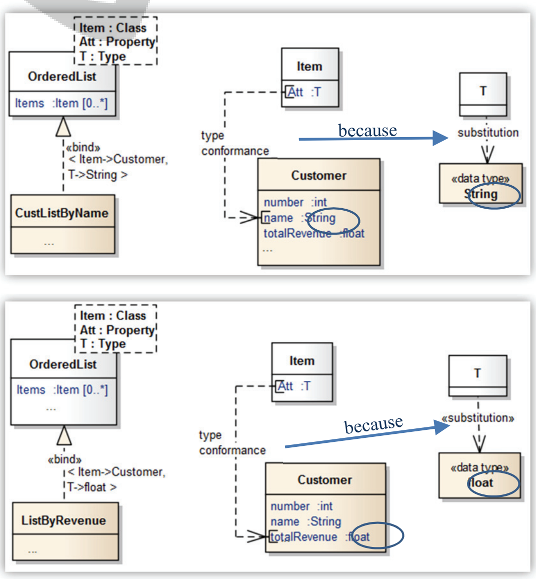


Figure 10: Type conformance when the type is substituted.

4.2 Multiplicity Conformance

UML imposes no constraints on template parameter substitution regarding the multiplicity of the involved elements. We propose the criteria of multiplicity conformance. This rule checks if two elements involved in a substitution are both single valued (upper multiplicity = 1) or both multivalued (upper multiplicity > 1) and, for the second case, both elements must have the same kind of ordering (ordered/not-ordered).

The single/multiple valued is important for computability because code that uses a multivalued element does it through flow control structures and operations that act upon collections of values (e.g.: foreach x in obj.feature, obj.feature.size(), etc.), while code using a single-valued element accesses it directly. Therefore, a computable piece of code that uses a single-valued element, becomes non-computable if that element is replaced by a multivalued one, and vice versa.

For multivalued elements, the ordered/not-ordered nature may also impact computability. On ordered elements one can apply operations that assume an ordering of values, such as the OCL operations *first ()*, *last ()*, and *at ()*.Calls to such operations will not compute if the ordered element is replaced by a non-ordered one. Similarly, some operations on non-ordered elements are not applicable to ordered ones, e.g. the OCL operation *intersection ()*.

Strictly, computability would be compromised only if the code of the template includes any of these operations depending on ordered/not-ordered. This aspect opens the possibility for establishing two levels of conformity enforcement – say, strict and flexible – a subject for future discussion.

It is also worth explaining that the reason why this conformance criteria doesn't take into account concrete values of multiplicity, other than 1 and *, is because it is considered a matter of semantic equivalence, not a requisite for computability. Albeit a legitimate concern, it is also postponed for future discussion.

4.3 Contents Conformance

This conformance criterion applies to template parameters that expose namespaces – namely: classes, associations, operations, packages, and all other constructs subclassifying *Namespace* in the UML metamodel (see (OMG 2012)). This rule is meant to certify that a substituting element (e.g., a class) contains substituting elements (e.g., member attributes) to all members that the template assumes there are in the parametered element. For instance, if a template has a parameter-class and uses the attributes *a1* and *a2* of that class, then every substitute must also be a class with attributes *a1* and *a2* or some substitutes for these. For instance, recalling the template shown in Figure 3, the class substituting *Item* must have an attribute *Name*.

The definition of this criterion requires the definition of another concept: *Implicit Substitution*. In the context of a bind, an element implicitly substitutes another if they are homonymous, functionally conform and the namespace of the former substitutes the namespace of the latter. In this definition, "homonymous" refers to having the same proper name, i.e., the elements have the same identification within the corresponding namespaces. For example, attributes *Item::Name* and *Person::Name* are properly homonymous. The same is true between the operations *Item::setName (String)* and *Person::setName (String)*. But not between *::setName (String)* and *::setName (String, String)*, because in UML an operation is identified by its name and signature. If two properly homonymous elements e^{T} and $e^{\odot}$ are also conformant in type, multiplicity, etc. (note the recursive definition) and the namespace of e^{T} is substituted by the namespace of $e^{\odot}$, then e^{T} is substituted implicitly by $e^{\odot}$. Notice that this definition is assuming that, even if the bind under consideration doesn't include an explicit substitution of e^{T} by $e^{\odot}$, such substitution will be made. For example, recalling Figure 5, previous statement implies that, even though the modeller doesn't specify the substitution of *Item*::*Name* by *Concept*::*Name*, such substitution is done. Thus, the concept of implicit substitution is an assumption regarding the semantics of the Bind relationship, regarding an aspect that UML's official documentation omits. Such assumption certainly deserves further discussion, yet postponed for another text. For the current purpose, implicit substitutions are assumed, just on the basis that the automatic substitution of an element by another with the same characteristics and name (or signature) is a reasonable option.

Thus, *Contents Conformance* is defined as: in the context of a template binding, namespace $NS^{\odot}$ conforms in contents to a namespace NS^{T} if every element in NS^{T} referenced by the template is substituted, explicitly or implicitly, by elements in $NS^{\odot}$.

This rule would detect problems such as the one previously shown in Figure 7. The substitution of *Item* by *Document* would be refused because those elements do not have conforming contents, since *Item::Name* is neither substituted nor homonymous of any attribute in *Document* (Figure 11).

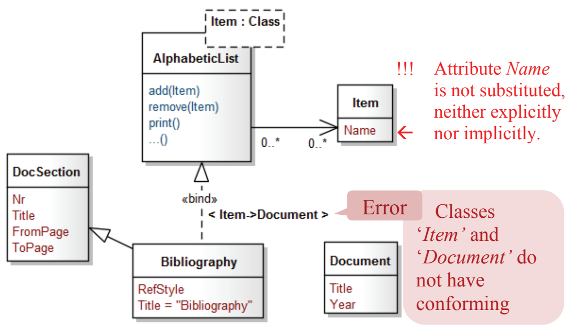


Figure 11: A violation of contents conformance.

The contents conformance requirement assumes two particular forms:

- A corollary named *Membership Conformance*, preferable to the general rule in certain, well-known situations;
- A specialisation applicable to operations, named *Signature Conformance*;

These more specific rules are analysed in the following subsections.

4.3.1 Membership Conformance

Erroneous binds such has the one previously show in Figure 9 would be prevented by the contents conformance rule. In that figure, *Person* will not be accepted as a substitute for *Item* because its contents don't fully substitute those of *Item* used by the template. Yet, the inadequacy of *Person* would be reported in a more specific way rephrasing that violation of contents conformance as: the substitution of *Item* by *Person* is not possible because one of *Item*'s attributes, *Name*, is not substituted by an attribute of *Person*. Or the problem could be imputed to the substitution of *Name* by *Code*: *Code* cannot substitute *Name* because its owning class does not substitute *Name*'s owning class. This last report exemplifies the application of the corollary *Membership Conformance*, defined as:

An element $e^{\oslash}$ conforms in membership to an element e^{T} if at least one of its namespaces substitutes one of e^{T}'s namespaces, either explicitly or implicitly.

Membership conformance is a sub-rule of (part of) contents conformance. It assesses the adequacy of a namespace's member instead of the adequacy of the namespace as a whole.

In the definition, the use of the plural "namespaces" is because an element may be inherited or imported and, consequently, be a member of several namespaces. Membership conformance is satisfied if the namespace substitution required by the rule occurs for any of these multiple namespaces, a detail that is not so apparent in the general rule (contents conformance). This is explained bellow.

Figure 12 shows a situation of membership conformance, involving inheritance. Since *Person* inherits *Name*, this attribute is member of *Entity* and *Person*, its namespaces. It is required that any of these classes substitutes *Item* in order to have membership conformance between *Att* and *Name*. It is so indeed (*Person* substitutes *Item*).

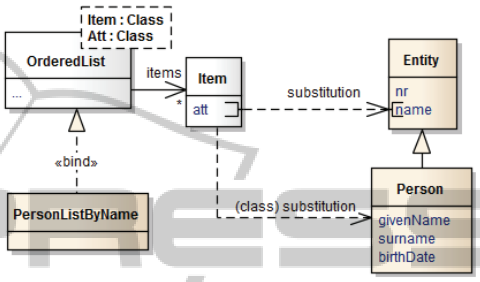


Figure 12: Membership conformance and inheritance.

An element that is member of a template also acquires namespaces by means of bind relationships. This is due to the fact that the semantics of the bind relationship includes a generalization (see Figure 5). Consequently, every element bound to a template becomes namespace of any non-private member of that template. Figure 13 shows a situation where membership conformance verifies, involving a bind. In that case, *Aged::Date* may substitute *Item::Att* because that attribute is member of Person and this class substitutes *Item*.

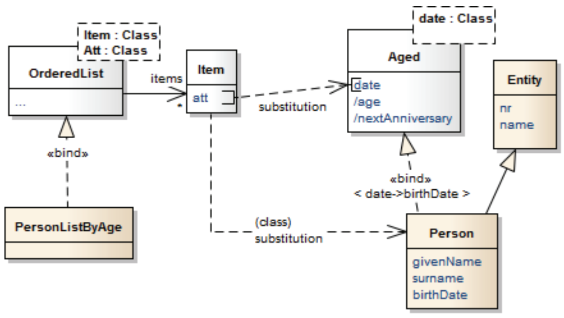


Figure 13: Membership conformance and binding.

As a guideline to choose whether a problem should be reported by contents or by membership conformance, it should be checked whether contents conformance doesn't hold due to a missing substitution or due to an incorrect substitution. For instance, in Figure 14 *att* may not be substituted by *wage* because that would lead to *PersonList* sorting

Person objects by wage, while not every object of *Person* has a *wage* attribute. This problem will be detected by the contents conformance requirement (the contents of *Person* do not fully substitute those of *Item*) as well as by membership conformance (none of the namespaces of *att* is substituted by a namespace of *wage*). This problem would be more appropriately reported as a membership problem: *wage* cannot substitute *att* because the class it belongs to (*Employee*) doesn't substitute the class *Att* belongs to (*Item*). In this situation contents conformance doesn't hold due to a bad substitution.

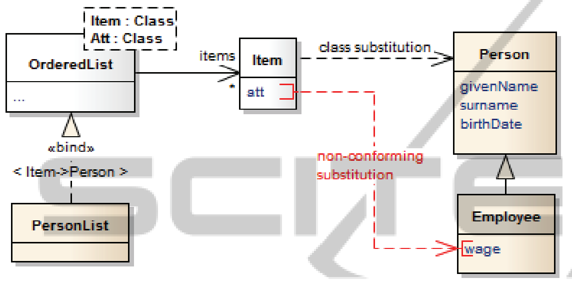


Figure 14: A situation where reporting by membership conformance is better than by contents conformance.

When contents conformance is not observed due to a missing substitution, explicit or implicit, of members of the parameter-namespace, then the problem is be better reported by the general rule (contents conformance). For instance, Figure 15 would raise an error message such as: *Person* cannot substitute *Item* because their contents do not conform. The problem could be further diagnosed, more specifically: *att* is not substituted. But this is not a violation of membership conformance. Such corollary is not even evaluable in the situation, since there is no prospective substitution for *att*.

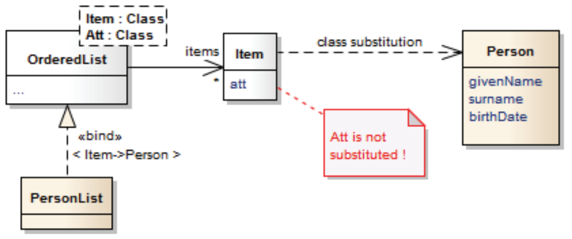


Figure 15: A situation where reporting by contents conformance is preferable to membership conformance.

4.3.2 Signature Conformance

In UML, an operation is a special case of namespace. The members of an operation are its parameters. Thus, contents conformance converts to signature conformance when it comes to operations.

Signature conformance checking is intended to assure that, when an operation foo_A $(p_{A1}, \ldots)$ is substituted by another foo_B $(p_{B1}, \ldots)$, the computability of calls 'foo_A (arg_A1, …, arg_An)' in a template is preserved when such calls are replaced by 'foo_B (arg_B1, …, arg_Bn)' in the bound element.

Since UML doesn't consider the concept of substitution between operation parameters, only implicit substitutions occur between elements of such kind. When foo_A $(p_{A1}, \ldots)$ is substituted by foo_B $(p_{B1}, \ldots)$, p_{A1} is implicitly substituted by p_{B1}.

The definition of implicit substitution also assumes a particular form, derived from the way operation calls in UML identify parameters when passing arguments: by their position in the signature of the operation. Therefore, for operation parameters, the definition of implicit substitution instead of saying "homonymous" says "in the same position in the signature".

Finally, conforming operations must have the same number of parameters. Indeed, while for other types of namespaces having more members than those that will participate in the substitution doesn't spoil computability, that isn't true for operations. In Figure 16, attribute *a3* in class *Cs* does not affect the conformance of *Cs* to *Cp*. On the contrary, parameter *p3* makes *OPs* non-conformant to *OPp*.

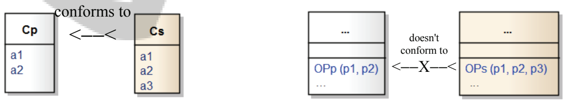


Figure 16: The number of operation parameters is relevant for functional conformance.

Since operation parameters are elements with type and multiplicity, conformance regarding those aspects is required. Additionally, to have two parameters functionally equivalent, they must have the same direction (in/out/inout/return), a third condition for conformance among such elements.

4.4 Staticity Conformance

Since static features are executed by the classifier and non-static by instances of the classifier, staticity clearly affects computability. Therefore, functional conformance between two features requires they are both static or both non-static.

4.5 Visibility Requirement

Finally, there is a requirement relevant to computability that doesn't involve the pair substitute/substituted, but rather the pair substitute/bound element. That's why it is not called "conformance".

An element may substitute a parameter only if that element is visible from the bound element.

This requirement is easily understood since the substitutions are done in the code of the bound element: since the bound element will refer to substitute elements, it needs visibility of such elements.

4.6 Computability Assurance

An element bound to a template is computable if the template itself is computable and if, for every parameter substitution, the substitute functionally conforms to the parametered element.

A formal demonstration is required to prove such statement. This can be done by demonstrating that every computable expression and statement in a template will be reproduced as a computable element if all substitutions verify the criteria for conformance. For lack of space, such demonstration will be provided in a future paper. Appealing to the reader's intuition, the following explanation is provided:

According to the semantics of the bind relationship, the template element will be equal to the bound element except at the points it references a substituted parameter. At the reproduction of such points, the bound element will be referencing the substitute. Let's call those "points of difference". If the template is computable, only at the points of difference the bound element could be non-computable. If every substitute functionally conforms to its parametered element, that substitute will:

- Be successfully used in contexts that are reproductions of its parameter's contexts;
- Respond successfully to services that are reproductions of its parameter's services;
- Yield results that are reproductions of its parameter's results.

Therefore, if at the points of difference the substitutes are doing well, the bound element is computable at those points and, consequently, fully computable.

5 RELATED WORK

Research aiming at improving the UML Template model is scarce. (Caron and Carré 2004) and (Vanwormhoudt et al. 2013) are the pieces of work most affine to the one presented in this paper.

Like current paper, (Caron and Carré 2004) also propose a set of well-formedness rules, additional to that of standard UML, aiming at strengthen the notion of template as a means to enforce the correctness of elements bound to a template. (Caron and Carré 2004) is not very specific on the level and/or kind of correctness that is ensured by the proposed set of constraints. If it were to ensure computability, it overlooks some important aspects, such as multiplicity, staticity, and visibility. There are also minor inaccuracies, probably by lapse (for instance, the imposition that a parametered element must be owned by the template).

(Vanwormhoudt et al. 2013) proposes the concept of Aspectual Templates (AT) to enforce structural conformance between a template and the model it is applied to. (Vanwormhoudt et al. 2013) states that ATs have only one parameter, which is a model, and defines a set of constraints to enforce structural conformance between the parameter and its substitute. Generally speaking, structural conformance has the same goal as functional conformance in current paper. But our concept is more complete and comprehensive. More complete because ATs omit some UML concepts and, by doing so, become too strict on the one hand and too indulgent on the other hand. For instance, by omitting inheritance ATs forbid substitutions by an inherited feature (too strict). By omitting multiplicity ATs allow a multivalued property be substituted by a single-valued one (too indulgent). Our approach is also more comprehensive because it works for any kind of templateable and parameterable element in UML. Finally, the *Apply* operation proposed by (Vanwormhoudt et al. 2013) is roughly the same as binding to a package-template.

Although with a goal different from current paper's, (France et al. 2004) also introduces a technique to validate structural conformance between a template and its bound elements. Although the proposed extension to UML put some added value in terms of expressiveness, the conformance verification method overlooks several aspects essential to computability, such as multiplicity and signature conformance.

Considering the field of Aspect Oriented Modelling, one can find plenty of methods with the same goal as current paper: how to obtain concrete, correct solutions from generic ones. Because those methods use approaches and formalisms other than UML templates, the comparison would be somewhat pointless. The only exception we are aware of is the Theme/UML approach (Clarke and Walker 2005), which uses UML package templates to model crosscutting functionalities. Theme/UML extends the concept of template to incorporate aspect-oriented capabilities. Although it supports the definition of parameters with owner-member relationships, which resembles contents/membership conformance in the current paper, it is not clear if substitute elements (which are also organized in owner-member relationships) are checked against parameters. For further exploration of the Aspect-Oriented Modelling field a good starting point could be the survey in (Wimmer et al. 2011).

6 CONCLUSIONS AND FUTURE WORK

The concept of functional conformance proposed in this paper has been experimentally applied to a reasonably large set of templates (aprox. 40) and application domains (12, some of them with alternative models). Such experiments showed a success rate of 100%, which provides some empirical evidence of the effectiveness of the approach. However, the authors believe that a more reliable demonstration should be provided. With that goal, a formal demonstration has been developed, to be published as soon as possible.

The aforementioned experiments also suggested that, when sorting out substitutes for a parameter, taking into account functional conformance may leverage automatic or semi-automatic substitution. Therefore, additionally to computability assurance, automatic binding is a potential benefit of functional conformance. This is a line of work to develop.

Another perception instilled by these experiments was that UML templates would better allow for greater flexibility. For instance, if a template is designed to work on an association it would be useful if one could use it on a chain of two connected associations.

REFERENCES

Caron, O. & Carré, B., 2004. An OCL formulation of UML2 template binding. In T. Baar et al., eds. *UML' 2004 — The Unified Modeling Language. Modeling Languages and Applications*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 27–40.

Clarke, S. & Walker, R.J., 2005. Generic Aspect-Oriented Design with Theme/UML. In *Aspecto-Oriented Software Development*. Addison-Wesley, pp. 425–458.

France, R.B. et al., 2004. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3), pp.193–206.

OMG, 2012. OMG Unified Modeling Language (UML), Superstructure, v2.4.1. Available at: http://www.omg.org/spec/UML/2.4.1/ (Accessed April 27, 2012).

Vanwormhoudt, G., Caron, O. & Carré, B., 2013. *Aspectual Templates in UML*, Available at: http://hal.archives-ouvertes.fr/hal-00846060.

Wimmer, M. et al., 2011. A survey on UML-based aspect-oriented design modeling. *ACM Computing Surveys*, 43(4), pp.1–33.

APPENDIX

OCL Formulation of Functional Conformance

Auxiliary definitions

```
context TemplateBinding
def: substituteOf (p: ParameterableElement)
   : ParameterableElement
   = self.parameterSubstitution
         .any (formal
               .parameteredElement = p)
         .actual
```

Type conformance

```
context TypedElement
def: typeConformsTo
       ( p: ParameterableElement,
         b: TemplateBinding) : Bool
   = let allTypes = p.type.allParents()
                     .including (p.type)
     in
     allTypes.forAll ( tp |
        let tpSubs = b.substituteOf (tp)
        in
          if tpSubs = null then
            self.type.conformsTo (tp)
          else
            self.type.conformsTo (tpSubs))
```

Multiplicity conformance

```
context MultiplicityElement
def: multiplicityConformsTo
       (p: MultiplicityElement) : Bool
   = self.upper = 1 and p.upper = 1
     or
    (self.upper > 1 and p.upper > 1 and
     self.isOrdered = p.isOrdered)
```

Contents conformance

```
context TemplateBinding
def: implicitSubstituteOf
      ( p: NamedElement) : NamedElement
  = let subsNs
    = p.elementNamespaces
      ->collect   (ns|   self.substituteOf
(ns))
      ->union (p.elementNamespaces
      ->collect (ns |
            self.implicitSubstituteOf
(ns)))
       ->asSet()->excluding (e | e = null)
    in
      if subsNs.isEmpty() then
        implicitSubstituteOf = null
      else
        subsNs.collect (members)->any (
          not   isDistinguishableFrom   (p,
ns))
```

```
context Namespace
def: contentConformsTo
     ( p: Namespace,
       b: TemplateBinding ) : Bool
  = let elemsNotSubstituted
      = p.member
        ->intersection
            (b.signature.template
              .usedElements)
        ->excluding (p |
            b.substituteOf (p) <> null)
        ->excluding (p |
            b.implicitSubstituteOf (p)
                                   <> null)
    in
      elemsNotSubstituted->isEmpty()
```

Membership conformance

```
context ParameterableElement
def: membershipConformsTo
       ( p: ParameterableElement,
         b: TemplateBinding ) : Bool
   = p.memberNamespace
     ->collect (ns | b.substituteOf (ns))
     ->intersects (self.memberNamespace)
```

Signature conformance

```
context Operation
def: signatureConformsTo
       ( p: Operation,
         b: TemplateBinding ) : Bool
   = (self.parameter->size
       = p.parameter->size) and
     Sequence {1..self.parameter->size}
        ->forAll ( i |
            self.parameter->at(i)
             .conformsTo (
               p.parameter->at(i), b))
```

```
context Parameter
  def: conformsTo
           ( p: Parameter,
             b: TemplateBinding ) : Bool
     = self.typeConformsTo (p, b) and
       self.multiplicityConformsTo (p) and
       self.direction = p.direction
```

Staticity conformance

```
context Feature
def: staticityConformsTo
        ( f: Feature ) : Bool
   = self.isStatic = f.isStatic
```

Visibility requirement

```
context TemplateableElement
def: hasVisibilityOf
          ( e: NamedElement ) : Bool
     = self.allNamespaces()
       ->first().hasVisibilityOf (e)
-- By default, an element forwards
-- the query to its closest namespace,
-- until it gets a namespace that
-- redefines this operation.
```

```
context Classifier
def: hasVisibilityOf
          ( e: NamedElement ) : Bool
   = if e = self then
       hasVisibilityOf = true
     elseif self.allParents().member
            ->includes (e) then
       hasVisibilityOf =
         (e.visibility <> #private)
     else
       hasVisibilityOf =
         (e.visibility = #public)
```

```
context Package
def: hasVisibilityOf
          ( e: NamedElement ) : Bool
   = if e = self or
        self.allOwnedMembers()->includes
(e)
     then hasVisibilityOf = true
     else hasVisibilityOf =
                   (e.visibility = #public)
```