Extending UML Templates Towards Flexibility (extended version)

José Farinha

ISTAR, ISCTE-IUL Lisbon University Institute, Lisbon, Portugal
jose.farinha@iscte.pt

Abstract. UML templates are generic model elements that may be instantiated as domain specific solutions by means of parameterization. Some of the elements in a template definition are marked as parameters, implying that these must be substituted by elements of the domain model, so to get a fully functional instance of the template. On parameter substitutions, UML enforces that the parametered element and its substitute must be of the same kind (both classes, both attributes, etc.). This paper shows that this constraint confines the applicability of templates and proposes an alternative that, by allowing substitutions among elements of different kinds, broadens that applicability. Cross-kind substitutions, however, require adequate semantics for the *Binding* relationship. Such semantics are proposed as model transformations that must complement the plain substitutions preconized by UML. Examples of such transformations are provided for activities in a template being expanded into a bound element.

Keywords: UML Templates, Genericity, Model Verification, Binding.

1 Introduction

Template is the construct in UML that brings into modelling the principles of Generic Programming (GP). In GP, program code is written abstractly, in terms of to-be-defined types, in order to factor out the commonalities of concrete solutions into general, replicable ones. Generic code can be instantiated multiply, wherever the addressed problem occurs and concrete types comply with the requirements of that code. In UML, generic model elements – *templates* – may be written in terms of a multitude of to-be-defined elements: types, properties, operations, packages, etc. These are said *parametered elements* or, simply, *parameters* (of a template). When instantiating a template, all of the parametered elements must be substituted by conforming elements of the application model. UML verifies the conformance between each parameter and its substitute imposing that they are of the same kind, i.e., have the same metaclass: a class must be substituted by a class, a property by a property, etc. Apart from that, UML (as to the current version, 2.5) barely imposes constraints on substitutions: the verification of the elements' types is the only relevant exception. Hence, most of the well-formedness verification of template instances is passed on to other constructs of the language: e.g., if a property is not an appropriate substitute, UML relies that

somewhere in the application model, any expression, message, action, etc. will fail in using that property and, by raising an ill-formation error, will prevent the substitution.

A similar strategy was used in C++ and it showed to lead to poor error reporting, a fact that instilled the introduction of C++ *Concepts* as a way to improve the validation of template instantiations [7]. Acknowledging that such reporting problems also occur in UML, [6] proposes an additional set of well-formedness rules to ensure that template parameters are properly substituted. With such constraints, a parameter's substitute is validated directly by the *Substitution* construct and incompatibilities are reported exclusively in terms of the substituted element, its substitute, and the *binding* relationship under consideration.

The current paper shows that such a set of constraints is, actually, sufficient to guarantee the well-formedness of template instances and that, therefore, the *same-kind* constraint may be relaxed. As such, *cross-kind* substitutions may occur – e.g., a property may be substituted by an operation, or vice-versa – granted that the substituted and substituting elements have compatible properties and that the semantics of the *Binding* relationship is enhanced with some simple model transformations. Since the *same-kind* constraint limits the applicability of templates, relaxing it will provide greater versatility to templates.

The structure of the paper is as follows: §2 illustrates the problem, showing through a simple example how the *same-kind* constraint restrains the application of templates; §3 presents a solution, by explaining the rationale behind it, the criteria that replace the *same-kind* constraint, and the additional semantics required so the Binding relationship deals adequately with cross-kind substitutions; §4 presents related work; §5 draws some conclusions and outlines future developments.

In diagrams in this paper, elements participating in a template definition are white-filled and elements of the application setting are wheat-filled.

2 A Motivating Example

To illustrate how the *same-kind* reduces the applicability of templates it will be used the simple template shown in Fig. 1.a along with a binding to it. *AlphabeticList* is a class template with two parameters: *T*, a class in the template space, and *Name* an attribute of that class. *AlphabeticList* embodies a generic solution for keeping a list of items ordered by some of those items' textual property. The semantics of the Binding relationship makes Fig. 1.(a) equivalent to Fig. 1.(b). Using OMG's UML terminology, the anonymous class *AlphabeticList <Document, Title>* represents the *expansion* of *AlphabeticList* into *Bibliography*, the element that bounds to the template.

Bounding to *AlphabeticList* will pose some difficulties if the bound element has the usual object-oriented encapsulation strategy based on getters and setters, as in Fig. 2. The problem is that with such binding, when processing the alphabetic ordering, the code of the resulting *AlphabeticList<Document, Title>* will try to access a private member of *Document*. Consequently, although the substitution of attribute *Name* by *Title* doesn't meet any impediments from UML template's validation rules, the validation of the resulting code will fail and the binding will not succeed.

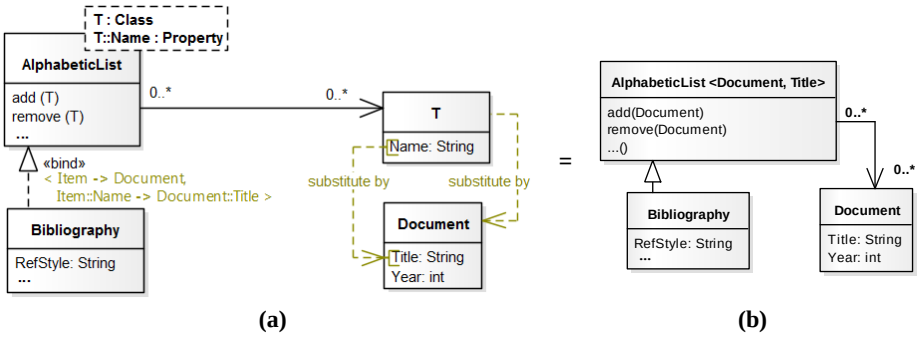

Fig. 1. Template *AlphabeticList* and a binding to it

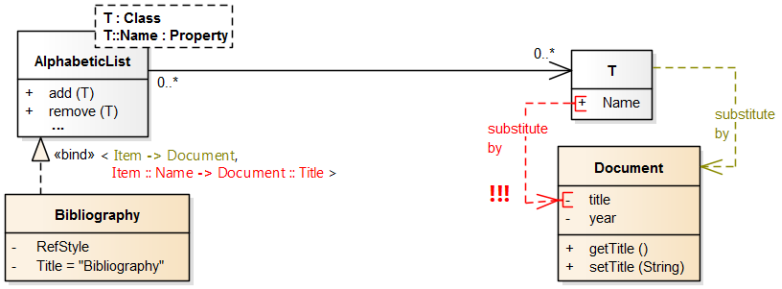

Fig. 2. A binding to *Alphabetic List* with bad-formation side-effects

To have our template applied to this target model, attribute *Name* would have to be substituted by operation *getTitle()*. However, such substitution is not allowed by UML, because *Name* and *getTitle()* are not of the same kind. It could be argued that this could be overcome defining an additional public attribute whose values are derived by the expression '= getTitle()'. Yet, such an attribute would be redundant. And it's not hard to foresee that, if the same approach is used as more and more templates are applied, domain classes would easily get overloaded with redundant features.

Another alternative could be put forth: *getName()* could be defined on *Item* and exposed as parameter, instead of *Name*. That would however lead to the same problem: if the application domain has only an attribute *Title*, instead of *getTitle()*, the template would not be applicable again. Only the definition of both *Name* and *getName()* in *Item*, along with two templates – one having *Name* as parameter and the other *getName()* – could provide applicability to both the application scenarios.

The template could also be of use in Fig. 3, so that each invoice has a list of *OrderItems*, sorted by the ordered products' names. However, such a binding would also cause errors in the expansion of the template's code into *Invoice*, because that code will try to access attribute *Name* on *OrderItem* objects. Once again, the definition of a derived attribute *Name* in *OrderItem* would be redundant. Only a third variant of the template will do: one that deals with scenarios were the ordering attribute is one association-end away from the listed objects. It might be becoming clear by now that, with this approach, templates tend to proliferate: another one would be required to deal

with ordering by an operation one association-end away, yet another to deal with an attribute two association-ends away, and so on. Like in Fig. 4.

It would be good if templates could be applied to different configurations of application models without a proliferation of features nor of template variants.

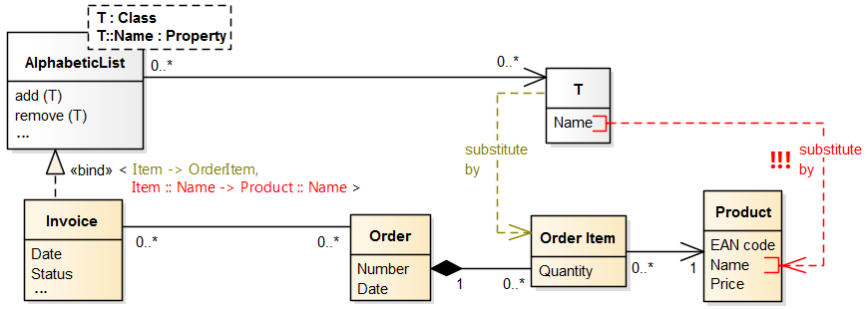

Fig. 3. A desirable but not viable instantiation of the template

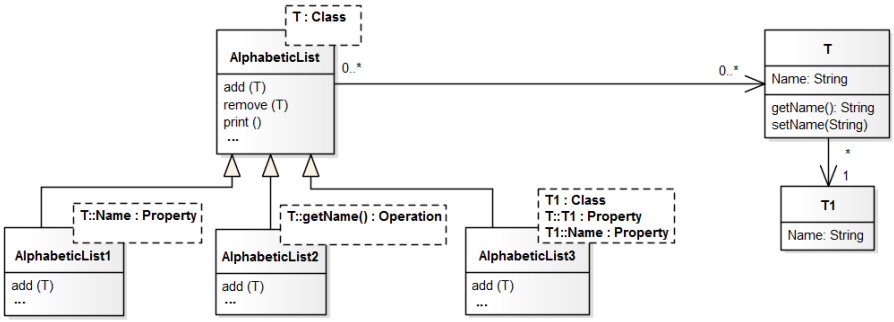

Fig. 4. Proliferation of template variants

3 The solution

In order to have a single template applied both to scenarios with properties (Fig. 1) and scenarios with operations (Fig. 2), such template would have to be defined generically in terms of a UML feature and have it exposed as a parameter. Since both properties and operations are features, both could be substitutes for that parameter. However, *Feature* is an abstract metaclass, so there is no way to have the afore mentioned feature. Hence, the template must be defined in terms of a property and have that property substituted by an operation, or vice versa. Furthermore, if cross-kind substitutions were allowed, the template could also be applicable to the scenario in Fig. 3, having the *OrderItem::Name* property substituted by the expression ".product.name". These examples suggest that relaxing the *same-kind* constraint is the way to go.

The *same-kind* constraint in UML 2.5 is imposed because of the bare semantics of the binding relationship: the body of the template is plainly copied to the bound element and, in the resulting copy, all references to parameters are replaced by refer-

ences to their substitutes. Since the new referencing elements are mere copies of their originals in the template, the elements they reference (the substitutes) must be of the same kind as the corresponding parameters. Therefore, cross-kind substitutions may be enabled only if the referencing elements undergo a transformation during a template expansion. Necessarily, one that replaces all dependencies on a parameter's metaclass by dependencies on its substitute's metaclass. E.g., substituting a property by an operation is possible, if all *ReadStructuralFeatureActions* referencing that property are transformed into *CallOperationActions* (which roughly corresponds to transforming "obj.aProperty" into "obj.anOperation( )", in textual programing). However, since not every kind of referencing elements may be converted into every other kind without loss of semantics, relaxing the *same-kind* constraint shouldn't mean allowing arbitrary substitutions, as shown in §3.2.

3.1 Bind Conformance

In [6] a concept named "Functional Conformance" was introduced to ensure the well-formedness of UML template instances. The concept is proposed as a set of metamodel constraints on template parameter substitutions. In the current paper the concept is extended to suit the new goals and renamed to *Bind Conformance* (BC)[1]. BC is presented in the next sections in a brief, informal way. The interested reader may find the corresponding formulations in [6]. In the following paragraphs, the prefix 'σ' is used to denote "substitute of".

Type conformance (Typ_{cnf}) is a criterion announced three-fold, considering a parametered element *e* and its substitute σ*e*:

(1) If a type *T* of *e* is not substituted, then σ*e* must have *T* as type;

(2) If a type *T* of *e* is substituted, then σ*e* must have σ*T* as type.

(3) If *e* has no type, then σ*e* must have no type as well.

UML only enforces (1) [8]. (3) is introduced in this paper because operations are considered type-conformant and void operations must be dealt accordingly. Typ_{cnf} applies to every element that may have a type, i.e., according to the UML's metamodel, *TypedElement*s and *Operation*s, but also to *Behaviors*, which may emulate a *type* metaproperty. Both *Operations* and *Behaviors* have their types derived from the type of their *return* parameters. The proposed metamodel is in Fig. 5.

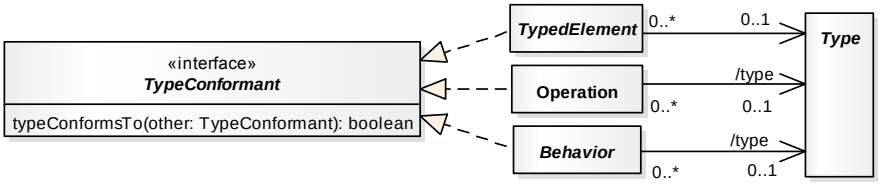

Fig. 5. Type conformance metamodel.

[1] The renaming is not a consequence of the new purposes, but to provide a more suitable name.

Subtyping conformance applies to classifiers and intends to preserve every *is-a* relationship from the template to the bound element. The definition is: if T is a subtype of T_{super}, then σT must be a subtype of σT_{super} or σT_{super} itself.

Two elements conform regarding multiplicity if they are both single-valued (multiplicities' upper bound = 1) or both multivalued (upper bound > 1) and, in the latter case, if they are both ordered or both not-ordered. **Multiplicity conformance** applies to UML's *MultiplicityElements* – Properties, Variables, Parameters, Connector ends, and Pins – as well as to *Operations*, *Behaviors* (these two's multiplicities are derived from their *return* parameters'), and *ValueSpecifications* – *Expressions*, *Instances*, and *Literals*.

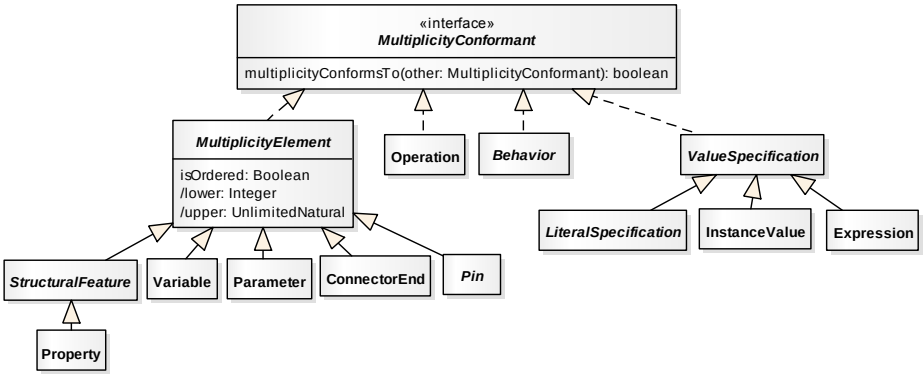

Fig. 6. Multiplicity conformance metamodel.

Members conformance (Mmb_{cnf}) applies to namespaces – Packages, Classifiers and Operations – and states that, in a binding, the namespace *Ns* is conformed by σNs if every member of *Ns* being used by the template is substituted by a member of σNs.

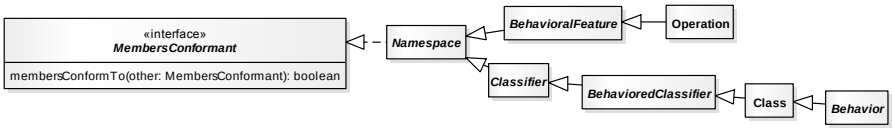

Fig. 7. Members conformance metamodel

Signature conformance (Sig_{cnf}) is a specialization of Mmb_{cnf} as applied to elements whose members are parameters, namely, *Operations* and *Behaviors* (*Activities* and *Interactions*). This criterion ensures that a substituting operation/behavior has a set of parameters compatible with that of the substituted. In this paper, Sig_{cnf} is also applicable to properties and expressions, which are considered pseudo-operations, having only one parameter, whose direction is *return*.

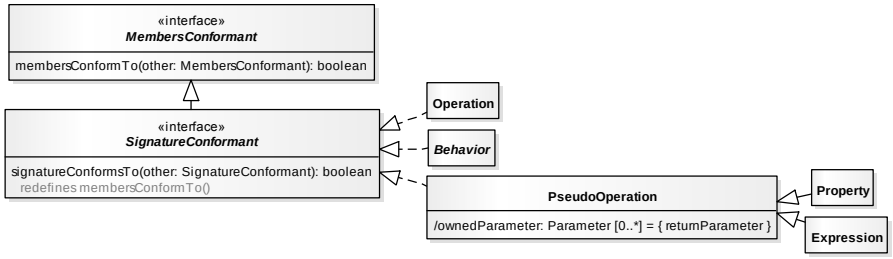

Fig. 8. Signature conformance metamodel

Direction conformance applies only to substitutions of operation and behavior parameters. It states that a substituting parameter must have the same a direction – *in*/*out*/*inout*/*return* – as the substituted.

Constancy conformance. Establishes that an element that is constant may only be substituted by another that is also constant, and a non-constant by another non-constant. Constant elements are: read-only structural features (properties), literal values, and every behavior or expression that exclusively references constant elements.

Staticity conformance states that static elements may only be substituted by another that is also static, and a non-static by a non-static. In UML, staticity is a characteristic exclusive to features. In this paper, it is extended to expressions and behaviors, which are considered pseudo-features. Expressions that reference the *self* variable are non-static; otherwise, are static. For behaviors, non-static are those that contain any action that uses *self* as the executor of a feature (textually, corresponds to those that call "self.aFeature") and all others are static.

Abstraction conformance applies only to classifiers and is already supported by UML 2.5. It states that a classifier that is not abstract may only be substituted by another that is also not abstract.

Enforcing Bind Conformance. BC must be enforced on template parameter substitutions. It may be defined as an invariant that merely calls *isConformedBy()* (Fig. 9). This operation is abstract in *ParameterableElement* and specialized on every subclass so to invoke the applicable conformance criteria. Fig. 9 shows how this is specialized for operations. If a criterion applies to a parametered element but not to its substitute, this one's metaclass will not be redefining the corresponding *...ConformedBy()* operation and, therefore, this' default definition (in *ParameterableElement*) will return *false*, causing a conformance failure, as desired.

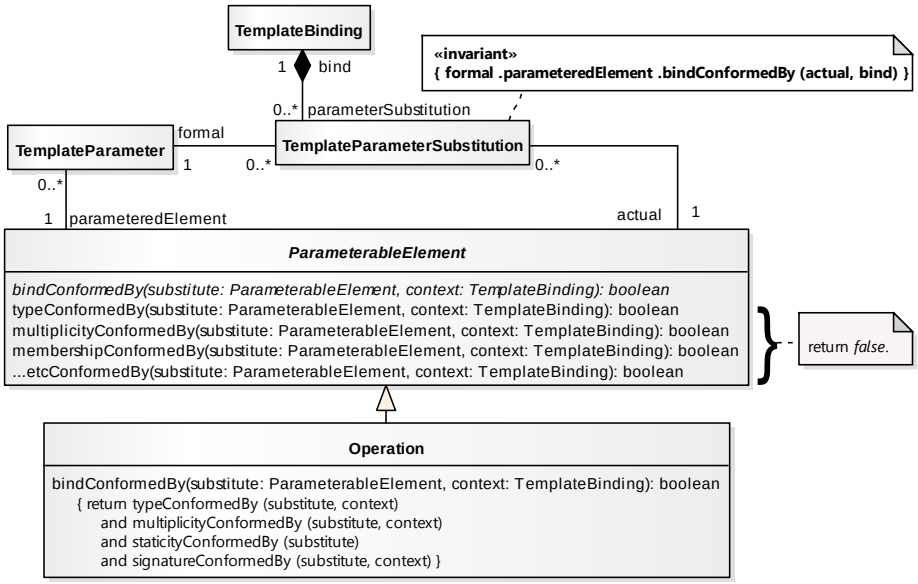

Fig. 9. Metamodel for bind conformance enforcement on operations

3.2 Possible metaclass combinations according to Bind Conformance

As mentioned above, relaxing the same-kind constraint doesn't mean allowing arbitrary substitutions. It is rather intuitive that some element kinds are incompatible from the start, independently of the particular elements playing parts in a substitution. E.g., if the parametered element is a property, a package is not certainly a suitable substitute. Therefore, it is necessary to have a rationale that, for each element kind, allows eliciting the kinds whose elements are prospective substitutes. In this paper, such rationale is built on the proposition put forth in [6] and demonstrated in [5]: for substitution purposes, the compatibility of two model elements depends on their kinds and on some of their syntactic properties – e.g., their types, multiplicities, signatures, etc. In fact, an operation with one parameter will not be properly substituted by an operation with two mandatory parameters, in spite of both being operations. Likewise, an operation with mandatory parameters will not work in contexts where a property is used. As substitutes of a property, eligible operations are only those with no mandatory parameters and whose return type conforms to the property's type. That's to say that, although the constraint on the kind of elements may be worked around, the syntactic compatibility doesn't. Hence, the need to preserve the syntactic compatibility is the basis of the afore mentioned rationale. This is the approach used in this paper.

Hence, the proposition is the following: a kind may be substituted by a different one if the latter's syntactic properties include all of the former's, allowing for the full verification of the syntactic compatibility between elements of these kinds. E.g., if the substituted element is an attribute, the relevant syntactic properties are: membership, type, multiplicity, visibility, constancy (constant/not constant), and staticity (static/not

static). Every element having such syntactic properties is a prospective substitute for an attribute. Hence, *Operation* – characterized by membership, signature (which includes type), multiplicity, visibility, constancy, and staticity – is a kind compatible with *Attribute*. Oppositely, *Package* is not compatible with *Attribute*, because packages are not characterized by type, multiplicity, constancy, and staticity.

Since the syntactic compatibility problem was already addressed in [6], which provided a solution in the form of a set of validation rules, the aforementioned proposition may be formulated adding a twist to [6]. Because [6] assumes that the same-kind constraint is in place, it only checks that every syntactic characteristic of the substitute is conformant to that characteristic of the substituted. For the current purposes, conformance must also be enforced the other way around: the substituting element must conform to *all of the substituted*'s syntactic characteristics. I.e., well-formedness constraints in [6] must be surjectively applied from the substituted to the substituting element.

The well-formedness rules proposed in [6] are extended in this paper in order to get as much benefits as possible from cross-kind substitutions. I.e., considering that the more flexible the better, the applicability of each criterion in [6] is extended to as many kinds as possible, given that it doesn't lead to misrepresentations of the associated concepts. That's to say that, although originally (in [6]) each criterion only applied to the kinds having the syntactic property under consideration, they are now extended to other kinds that are able to emulate such syntactic property. E.g., in [6] $\mathrm{Sig}_{\mathrm{Cnf}}$ applies only to operations. Herein, it will also apply to properties, since these can be seen as elements with a single *return* parameter. Likewise, formerly Typ_{Cnf} only applied to instances of *TypedElement*, which is not the case of operations. Herein, Typ_{Cnf} also applies to operations, since these have a *type* metaproperty, derived from the type of the *return* parameter. Such extensions maximize the compatibility of kinds, without eroding any well-formedness guarantees and semantics previously existing.

In Table 1 metaclasses are related to their associated conformance criteria. Elements of a metaclass may substitute elements of another if the former's conformance criteria include the latter's. Therefore, the allowed substitutions are shown in Table 2.

Table 1. Parameterable elements vs conformance criteria

		Conformance Criteria									
		Type	Multiplicity	Members	Membership	Signature	Direction	Staticity	Abstraction	Constancy	Visibility
Metaclass	Package			✓	✓						✓
	Classifier			✓	✓				✓		✓
	Behavior	✓	✓	✓	✓	✓		✓	✓	✓	✓
	Property	✓	✓		✓	✓		✓		✓	✓
	Operation	✓	✓	✓	✓	✓		✓		✓	✓
	Value Spec.	✓	✓		✓	✓		✓		✓	✓
	Parameter	✓	✓		✓		✓	✓			✓
	Variable	✓	✓		✓			✓			✓

Table 2. Compatible bind conformance criteria

	⌐>	Substituting							
		Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Substituted	Package	B	B						
	Classifier		B	B					
	Behavior			B	B	B	B		B
	Property			B	B	B	B	B	B
	Operation			B	B	B	B	B	B
	Value Spec.			B	B	B	B	B	B
	Parameter							B	
	Variable						B	B	B

Clearly, Table 2 elicits some bogus compatibilities. For instance, in no way a property could be substituted by a parameter. Such situations will be filtered out by recognizing that, although those kinds of element use the same criteria, some of these criteria are impossible to hold between them. This happens regarding membership conformance. A pre-requisite for two elements conform in membership is that they share at least a common kind of elements they could have as namespaces. E.g., since a parameter may only have an operation as namespace and a classifier may not have an operation as namespace, none of them could possibly conform to the other. Therefore, a grid of possible membership conformance must be superimposed to the previous grid. For such purpose, firstly, all parameterable elements must be associated to the metaclasses whose instances are allowed as their namespaces. This is shown in Table 3. Only parameterable elements that have at least a common metaclass for namespace (a common line marked with '::' in that table) may succeed in membership conformance. These combinations are shown in Table 4. The superposition of Table 3 with

Table 4 is shown in Table 5. Only those combinations that have both “B” and “M” may succeed in bind conformance, which are shown in Table 6.

Table 3. Allowed membership relations

		Members							
		Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Namespaces	Package	::	::	::					
	Classifier		::	::	::	::	::		
	Behavior		::	::	::	::	::	::	::
	Operation							::	

Table 4. Possible membership conformance

	⊏>	Substituting							
		Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Substituted	Package	M	M	M			M		
	Classifier	M	M	M	M	M	M		
	Behavior	M	M	M	M	M	M	M	
	Property		M	M	M	M	M		
	Operation		M	M	M	M	M		
	Value Spec.	M	M	M	M	M	M		
	Parameter			M				M	
	Variable								M

Table 5. Compatible conformance criteria plus possible membership conformance

	⊏>	Substituting							
		Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Substituted	Package	BM	BM	M			M		
	Classifier	M	BM	BM	M	M	M		
	Behavior	M	M	BM	BM	BM	BM	M	B
	Property		M	BM	BM	BM	BM	B	B
	Operation		M	BM	BM	BM	BM	B	B
	Value Spec.	M	M	BM	BM	BM	BM	B	B
	Parameter			M				BM	
	Variable						B	B	BM

Table 6. Possible cross-kind substitutions

Substituted \ Substituting	Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Package	↴	↴						
Classifier		↴	↴					
Behavior			↴	↴	↴	↴		
Property			↴	↴	↴	↴		
Operation			↴	↴	↴	↴		
Value Spec.			↴	↴	↴	↴		
Parameter							↴	
Variable								↴

3.3 Enhanced Semantics for the Binding Relationship

As mentioned before, the possibly of cross-kind substitutions relies on enhancing the semantics of the *Binding* relationship. In UML (v2.5), such semantics define the following about substitutions: when expanding the template body into the bound element body, every reference to a parametered element must be replaced by a reference to that element's substitute. For cross-kind substitutions, such plain replacements are not enough, and must be complemented with model transformations. For every reference to a parametered element, a transformation must convert the referencing element to another that adequately references the substituting element, taking into account this one's metaclass. Such transformations must replace all dependencies on the substituted element's metaclass by dependencies on the substituting's metaclass.

These transformations are specific to every combination of a referencing element's metaclass with a substituting element's metaclass. Consequently, there are plenty of them. For space reasons, this paper only exemplifies such transformations for a small set of referencing elements out of the UML's Activity formalism.

Transforming structural feature actions. With cross-kind substitutions enabled, the intended binding of Fig. 2 would succeed if *T::Name* is substituted by *Document::getTitle ()*. In activities owned by *AlphabeticList* – such as those defining the behavior of this class' operations – read accesses to *T::Name* are done through *Read Structural Feature* actions. These actions need to be transformed, and not merely copied, when that template is expanded into *Bibliography*. Such transformation is simply a conversion to *Operation Call* actions, as shown in Fig. 10.a.

Regarding the binding in Fig. 3, it would succeed if property *T::Name* is substituted by expression "product.name". Being this expression is owned by *OrderItem*, "product" will be interpreted as the association-end leading to class *Product* and, therefore, the expression will be successfully computed by *OrderItems*. Similarly to the previous example, actions reading *T::Name* in the template must be converted to *Specification Value* actions in the bound class (Fig. 10.b).

With the approach being proposed, elements that write to properties may not be converted. Therefore, properties being written to by the template are not eligible for cross-kind substitutions. This is a limitation of the current approach.

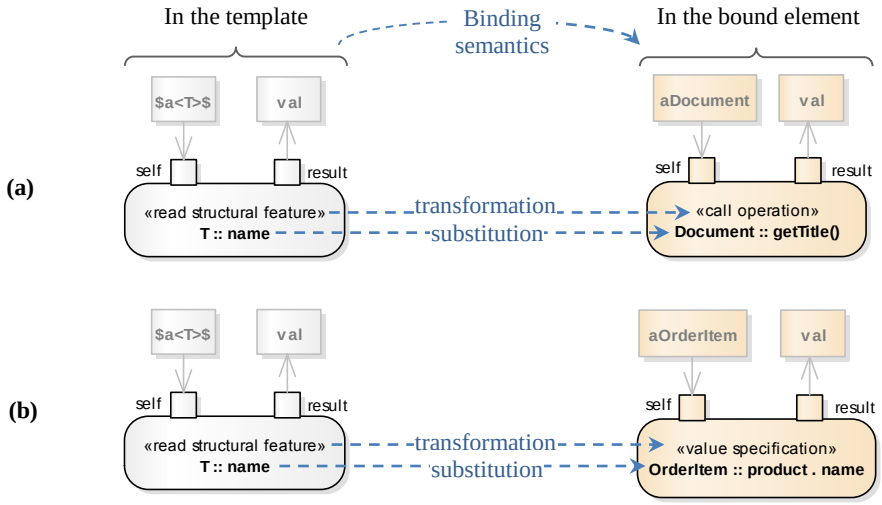

Fig. 10. Examples of the expansion of cross-kind substitutions

Transforming operation calls. Fig. 11 shows a template with three parameters: class *T* and two operations of that class, *value()* and *getValue()*. This figure shows also a binding to that template, which includes two cross-kind substitutions: operation *value()* by property *price*, and operation *getValue()* by activity '*price=*'. The '*price=*' activity is defined inside *Product* (as shown by the nesting relationship) and it may be no more than a wrapper for a *write structural feature* action. Such activity is required because actions are not parameterable elements in UML and, therefore, cannot participate in substitutions. For convenience, the specification of such action could be automatically generated from the textual declaration '*Item::setValue() -> price=*'. The transformations needed for expanding these substitutions must convert *Operation Call* actions to *Read Structural Feature* actions and *Activity Call* actions, respectively.

In order to take into account de effect of discounts on the price, operation *value()* could be substituted by an expression such as '*price * (1 –discount)*', which would require transformations into *Value Specification* actions.

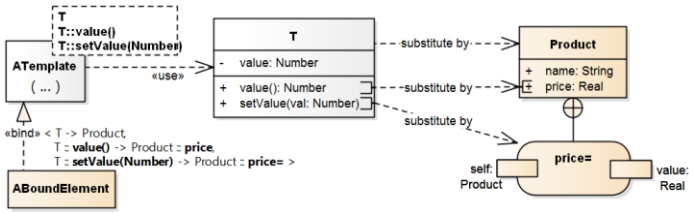

Fig. 11. Example of cross-kind substitutions

4 Related Work

Research on UML templates is scarce. Pieces of work with approaches to ensuring conformance in template binding similar to the one in this paper are [3], [2], and [11]. None of them aim at flexibility, as conformance validation is strict regarding the kind of elements. To our knowledge, the current paper is the first attempt to introduce flexibility in UML template binding.

In the GP field, *Concepts* are used to impose requirements on template arguments [4]. In Java Generics and in C# Templates, *Concepts* are specified through interfaces, the same approach as that of UML's *constraining classifiers*, having the same limitations. [9] and [7] are proposals for introducing *Concepts* in C++, and are the approaches that most resemble UML Templates with BC. A *concept* definition in a C++ template plays the same role as an element exposed as a parameter of an UML template, if BC is enforced. The advantage of UML+BC is that no additional constructs are required: *concepts* are modelled by ordinary classes, packages, operations, etc.

In the *Design Pattern* (DP) field, most of the approaches to DP instantiation and/or recognition acknowledge that a DP instance may not strictly mimic the DP model. E.g., GENBF, an extension to BNF for the specification of DPs, proposed in [1] and [12], considers that a relationship in a DP may be instantiated as a relationship chain. Mapping to the current paper's approach, that would correspond to substituting a property by a conformant dot-expression. [10] proposes an approach to formalizing DPs that combines conformance with flexibility, like the current paper, although in a different formalism.

5 Conclusions and Future Work

So far, the idea put forth in this paper was only experimented empirically, mostly using a set of templates dedicated to data validation. Such templates are mostly composed of generic data structures, constraints on that data, and some auxiliary operations. Their intent is to model at a high level of abstraction commonly observed data validation rules, and having them applied back to concrete settings. The observed results showed that cross-kind substitutions lead to a significant saving in the number of templates. Mainly, the substitution of properties by expressions allows that every single template becomes applicable to a several settings that otherwise would require multiple variants of the template. However, a future study for assessing the effect on the total complexity (of a template + bindings to it) seems advisable, since a decrease in a template library's complexity comes with an increase in the bindings' to it (because substitutions by expressions become more complex). The effect on the maintenance of bindings is also uncertain and should be evaluated as well.

References

1. Bayley, I., Zhu, H.: Formal specification of the variants and behavioural features of design patterns. J. Syst. Softw. 83, 2, 209–221 (2010).
2. Caron, O., Carré, B.: An OCL formulation of UML2 template binding. UML' 2004 — Unified Model. Lang. Model. Lang. Appl. 3273, 27–40 (2004).
3. Cuccuru, A. et al.: Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. Model Driven Eng. Lang. Syst. 12th Int. Conf., Model. 2009, Denver, CO, USA, 2009. Proc. 5795, 644–649 (2009).
4. Dehnert, J., Stepanov, A.: Fundamentals of Generic Programming. Int. Semin. Generic Program. Dagstuhl, Ger. 1998, Sel. Pap. 1766, 1–11 (1998).
5. Farinha, J.: A Demonstration of compilability for UML template instances. In: Procs. of the 4th Int. Conf. on Model-Driven Engineering and Software Development (Modelsward 2016). pp. 397–404 ScitePress, Rome, Italy (2016).
6. Farinha, J., Ramos, P.: Computability Assurance for UML Template Binding. In: Model-Driven Engineering and Software Development: 3rd Int. Conf., MODELSWARD 2015, Selected Papers. pp. 190–212 Springer (2015).
7. Gregor, D. et al.: Concepts: Linguistic Support for Generic Programming in C++. In: Procs. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA. pp. 291–310 ACM (2006).
8. OMG: OMG Unified Modeling Language, version 2.5. (2015).
9. Siek, J.G. et al.: Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC SC22/JTC1/WG21. (2005).
10. Soundarajan, N., Hallstrom, J.O.: Precision, Flexibility, and Tool Support: Essential Elements of Pattern Formalization. In: Taibi, T. (ed.) Design Pattern Formalization Techniques. pp. 280–301 IGI Global (2007).
11. Vanwormhoudt, G. et al.: Aspectual templates in UML. SoSym. (2015).
12. Zhu, H., Bayley, I.: An Algebra of Design Patterns. TOSEM, ACM Trans. Softw. Eng. Methodol. 22, 3, 23:1–23:35 (2013).