Extending UML Templates Towards Flexibility

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

The *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*: 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 substitute's type and of compliance to constraining classifiers (when specified) are the only relevant exceptions [8]. But these fall short in fully verifying the adequacy of a substitute, since the former only cares for types (e.g., multiplicities and visibilities are not checked) and the latter only applies to classifier parameters (not to parameters that expose properties or operations, for instance). Truly, 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 for a parameter, 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 constraints 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. The proposition put forth is that *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 are compatible, according to [6];
- The semantics of the *Binding* relationship is enhanced with model transformations that convert the way a substituted element is referenced to the way its substitute is referenced. That is to say that, if cross-kind substitutions take place, the body of the template may no longer be merely copied into the bound element. Instead, it must undergo a transformation. E.g., if a property is substituted by an operation, property-accessing elements must be transformed to operation-calling elements.

If such transformations could be applied in a systematic way, the *same-kind* constraint could be dropped from the UML metamodel. Since that constraint limits the applicability of templates, removing it provides greater versatility to templates.

Work undertaken so far suggest that the complexity of such transformations depends on the substitution capabilities being targeted. Yet, for a limited set of such capabilities, those transformations are derivable simply from the substituted and the substituting kinds. Therefore, it has been revealed that, up to some extent, cross-kind substitutions are viable in UML. For space reasons, the details on deriving transformations are postponed to a future paper. Current paper sets out the proposed solution showing the criteria that replaces the *same-kind* constraint and providing a glimpse of the transformation process through some intuitive examples.

The structure of the paper is as follows: §2 illustrates the problem, showing 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 in Fig. 1.a along with a binding to it. *AlphabeticList* is a class template with two parameters: type *T* and *Name*, a property of that type. *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.

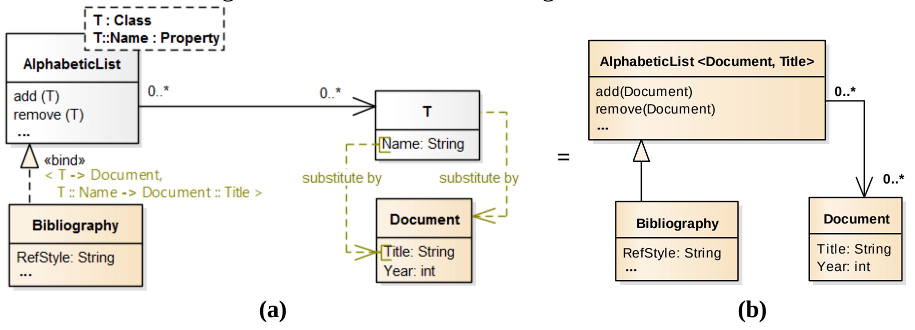

Fig. 1. Template *AlphabeticList* and a binding to it

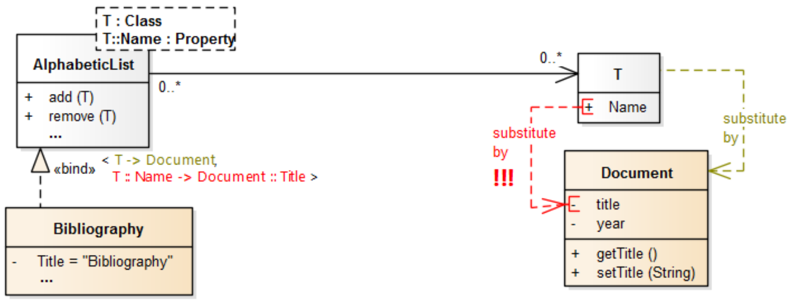

Fig. 2. A binding to *Alphabetic List* with bad-formation side-effects

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 *T* 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 *T*, 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. The definition of a derived attribute *Name* in *OrderItem* would solve the problem but, once again, it 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.

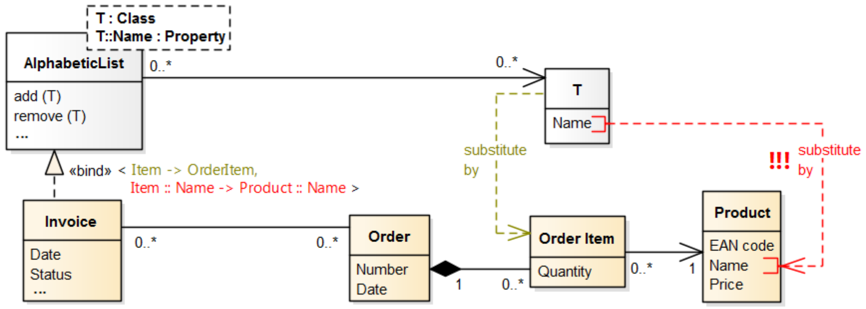

Fig. 3. A desirable but not viable instantiation of the template

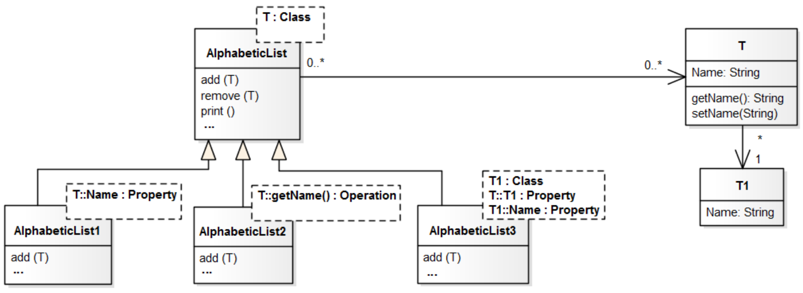

Fig. 4. Proliferation of template variants

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 references 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 meta-model 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 ($\mathbf{Typ_{Cnf}}$) 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.

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

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., UML's *TypedElement*s and *Operation*s.

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.

Multiplicity conformance states that 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.

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.

Signature conformance (*Sig_{cnf}*) ensures that a substituting operation/behavior has a set of parameters compatible with that of 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.

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.

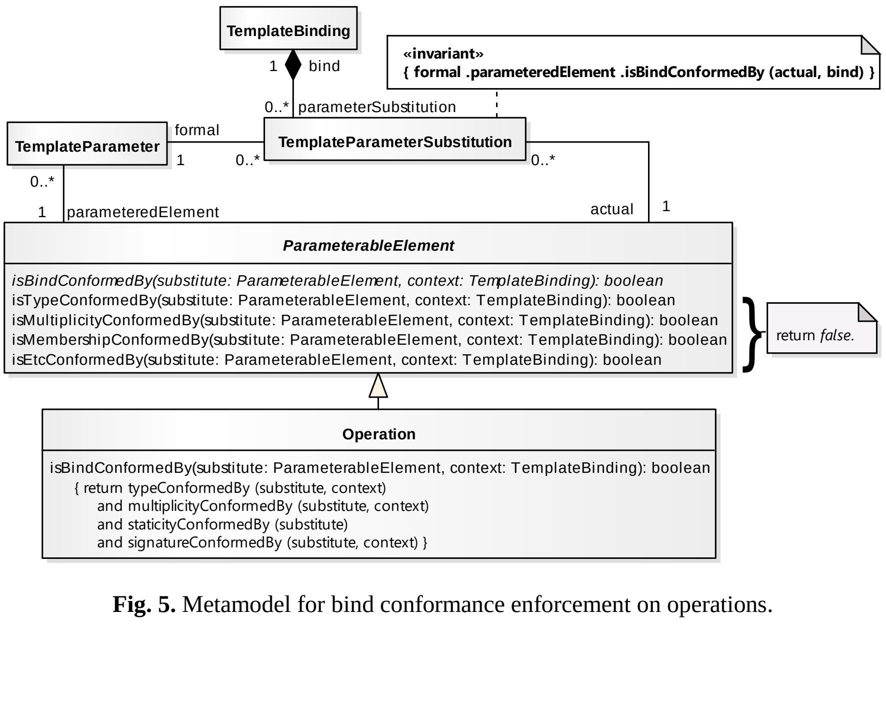
TemplateBinding
1
bind
«invariant»
{ formal .parameteredElement .isBindConformedBy (actual, bind) }
0..* parameterSubstitution
TemplateParameter
formal
1
0..*
TemplateParameterSubstitution
0..*
0..*
1 parameteredElement
actual
1
ParameterableElement
isBindConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
isTypeConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
isMultiplicityConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
isMembershipConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
isEtcConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
return false.
Operation
isBindConformedBy(substitute: ParameterableElement, context: TemplateBinding): boolean
{ return typeConformedBy (substitute, context)
and multiplicityConformedBy (substitute, context)
and staticityConformedBy (substitute)
and signatureConformedBy (substitute, context) }

Fig. 5. Metamodel for bind conformance enforcement on operations.

Enforcing Bind Conformance. BC must be enforced on template parameter substitutions. It may be defined as an invariant that merely calls *isBindConformedBy()* (Fig. 5). This operation is abstract in *ParameterableElement* and specialized on every subclass so to invoke the applicable conformance criteria. Fig. 5 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 *is<...>ConformedBy()* operation and, therefore, this' default definition (in *ParameterableElement*) will return *false*, causing a conformance failure, as desired.

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 allows eliciting what kinds are compatible. For space reasons, such rationale is not shown in this paper. The interested reader must refer to [5], where Table 1 is deduced.

Table 1. Possible cross-kind substitutions

		Substituting							
		Package	Classifier	Behavior	Property	Operation	Value Spec.	Parameter	Variable
Substituted	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 from the UML's Activity formalism.

Transforming Structural Feature (SF) actions. With cross-kind substitutions enabled, the intended binding of Fig. 2 succeeds if *T::Name* is substituted by *Document::getTitle ()*. If the *AlphabeticList*'s methods are defined as UML activities, read accesses to *T::Name* are done through *Read SF* actions. These need to be transformed, and not merely copied, when that template is expanded into *Bibliography*: *Read SF* actions must be converted to *Operation Call* actions, as in Fig. 6.a.

The binding intended in Fig. 3 succeeds if property *T::Name* is substituted by expression "product.name". Since this expression will be applied to *OrderItems*, "product" will be interpreted as the association-end leading to class *Product* and, therefore, the whole expression will compile successfully. Hence, actions reading *T::Name* in the template must be converted to actions that process expressions in UML, *Value Specification* actions (Fig. 6.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.

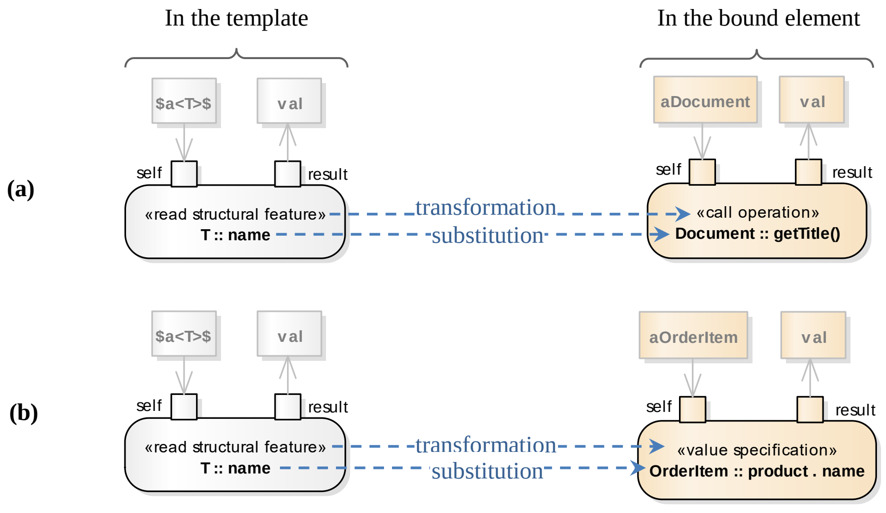

Fig. 6. Examples of template expansion with cross-kind substitutions

Transforming operation calls. Fig. 7 shows a template with three parameters: class *T* and two operations of that class, *value()* and *setValue()*. This figure shows also a binding to that template, with two cross-kind substitutions: operation *value()* by property *price*, and operation *setValue()* by activity '*price=*'. The '*price=*' activity is defined inside *Product* (as shown by the nesting relationship) and it may be simply a wrapper for a *Write SF* action. For convenience, such activity could be automatically generated from the textual declaration '*T::setValue() -> price=*'. The transformations processing these substitutions must convert *Operation Call* actions to *Read SF* actions and to *Activity Call* actions, respectively.

As a further example, operation *value()* could be substituted by an expression such as '*price * (1 –discount)*', which would require transformations to *Value Spec* actions.

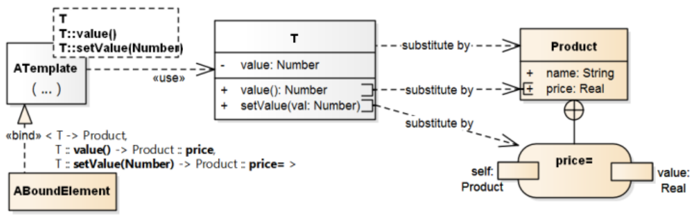

Fig. 7. Substituting operations by properties and activities

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], but none of them propose conformance on behalf of flexibility. 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 most of the languages (e.g., Java, C#, and Scala), *Concepts* are implemented through bounded polymorphism, the same approach as UML's *constraining classifiers*. When imposing that a type parameter subtypes a bounded type, these languages prevent cross-kind substitutions. [9] and [7] are proposals for introducing *Concepts* in C++, and are the approaches that most resemble UML Templates with BC: a C++ *Concept* corresponds to an element exposed as parameter in UML, if BC is enforced. However, none of these proposals capitalize conformance as a means to flexibility.

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, signifying some level of flexibility. E.g., [1] 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 a formalism for DPs that combines conformance with flexibility, like the current paper, but not for UML.

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. 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 several settings, that would otherwise 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' complexity (substitutions by expressions are more complex). The effect on the maintenance of bindings is also uncertain and should also be evaluated.

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.C., Stepanov, A.A.: Fundamentals of Generic Programming. Generic Program. Int. Semin. Generic Program. Dagstuhl Castle, Ger. 1998, Sel. Pap. 1766, 1–11 (1998).
5. Farinha, J.: Extending UML Template Towards Flexibility (extended version). , Lisbon, Portugal (2016).
6. Farinha, J., Ramos, P.: Computability Assurance for UML Template Binding. In: Desfray, P. et al. (eds.) Model-Driven Engineering and Software Development: 3rd Int. Conf., MODELSWARD 2015, Revised Selected Papers. pp. 190–212 Springer International Publishing, Cham (2015).
7. Gregor, D. et al.: Concepts: Linguistic Support for Generic Programming in C++. In: Procs. 21st ACM SIGPLAN Conference 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. Softw. Syst. Model. (2015).