Manuscript

The final publication is available at Springer via:
http://dx.doi.org/10.1007/s10270-015-0463-3

To cite this paper:
G. Vanwormhoudt, O. Caron, and B. Carré. Aspectual Templates in UML. Enhancing the semantics of UML templates in OCL. *Software & Systems Modeling*, 16 (2) pp. 469-497, Springer, May 2017.

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Aspectual Templates in UML

Enhancing the semantics of UML templates in OCL

Gilles Vanwormhoudt[1,2], Olivier Caron[1] and Bernard Carré[1]

[1] CRIStAL, UMR CNRS 9189
University of Lille
France - 59655 Villeneuve d'Ascq cedex
e-mail: {Gilles.Vanwormhoudt, Olivier.Caron, Bernard.Carre}@univ-lille1.fr
[2] Institut Mines-Telecom

Received: date / Revised version: date

Abstract UML Templates allow to capture reusable models through parameterization. The construct is general enough to be used in many ways, ranging from the representation of generic components (such as Java generics or C++ templates) to aspectual usage, including pattern, aspect and view oriented modeling. We concentrate on this last usage and so called "Aspectual Templates" which require that parameters must form a model of systems in which to inject new functionalities. Starting from this strict constraint, we derive an in-depth semantic enhancement of the standard. It is formalized as a fully UML-compliant interpretation in OCL of the template construct and its binding mechanism. In particular this aspectual interpretation must be ensured in case of partial binding (not all parameters are valued). Partial binding of UML is a powerful technique which allows to obtain richer templates from the composition of other ones. As a major result, the present semantic enhancement is consistent with this capacity so that partial binding of aspectual templates produces aspectual templates. Finally, at an operational level, an algorithm for aspectual template (partial) binding operation is formulated and consequent reusable technology made available in EMF (Eclipse Modeling Framework) is presented.

Key words Model Templates, UML, OCL, Metamodeling, Aspects, Patterns, Template Composition.

1 Introduction

After being considered only as documentation for a long time, software models are nowadays first-class objects. The MDA methodology (Model Driven Architecture [1]) identified the need to separate platform-independent modeling choices from platform-dependent ones in order to facilitate subsequent software generation, with respect to "vertical" transformation chains. Then MDE (Model Driven Engineering [25]) generalized the approach. It upgraded the status of models, from components dedicated to MDA steps, to full first-class software objects that are reusable and composable. The challenge is to facilitate the reuse of technology independent design efforts and logics in a productive and safe manner.

Once it was clear that software models could be isolated and composed, powerful techniques inspired from the programming world were considered to increase their reusability, such as model parameterization. The UML technology [43] contributes a lot to this trend while trying to capture common concepts and techniques, specifically through its concept of "model template". The ambition is to support much of MDE practices which call for model parameterization [44]. They are of two kinds. First kind is the representation at a model level of generic software components, such as C++ templates or Java generics, and then their instantiation [4, 16, 17, 20]. Second kind is the specification of overall and reusable software dimensions and then their application to a system being in construction, mainly the way aspect-orientation did [26], leading to the notion of "aspectual templates" studied here. This is related to aspect-oriented modeling (AOM) [11,24,27,36,38,45,47], subject or view oriented modeling including Catalysis frameworks [9,11,20,33,46] and pattern guided design [13, 39].

UML templates allow to represent model schemas where some of their ingredients are listed as parameters. Its specific "binding" relationship allows to specify how a model is related to a template through the substitution of its parameters. It is worth noting that parameters are only (meta-typed) individuals and form an unstructured set of model elements of the template. So that

the construct is general and permissive enough to render much of model parameterization needs, as mentioned above. But when aspectual usage is concerned, parameters have to specify a plain required model of systems in which to add the new functionalities. This superimposes to the standard construct that parameters must form a full well-formed model to which candidate models have to conform.

This leads to the definition of "aspectual templates" and the demonstration. As far as "parameters must form a model" was recognized as the initial requirement for aspectual interpretation of UML templates (a kind of postulate), subsequent reinforcement of their semantics can and must be derived, which consists in ensuring this requirement throughout definitions and composition mechanisms. Following the standard, the solution consists of constraining its metamodel thanks to the OCL formalism. It is worth noting that the standard being general enough as seen above, only refining its semantics in order to capture aspectual needs must be sufficient. As a result, this will be done in this paper so that aspectual templates are full UML templates (but not vice versa w.r.t. the openness of the standard specification to other template interpretations).

Moreover, UML templates allow partial binding where not all parameters are valued and must remain parameters in the resulting model, so that it is itself a template. The technique allows to obtain richer templates from the composition of other ones in a hierarchical way and then facilitates the constitution of "off-the shelf" model template libraries. In case of aspectual templates, following their proper constraint that "parameters must form a model", partial binding of aspectual templates must produce aspectual ones. We will see that the present semantic variation of the standard guarantees this property and it is a proof of its consistency.

After providing background on UML templates and their metamodel (Section 2), we present major existing works which call for aspectual interpretation of UML templates and identify the issues in Section 3. In Section 4 we show how UML template semantics can be enhanced to render aspectual one. The obtained engineering facilities and their properties are motivated through typical scenarios. After that (Section 5), the semantic enhancement in OCL of the UML template metamodel dedicated to aspectual templates is detailed. Section 6 specifically addresses the issue of partial binding between templates and presents a composition strategy which is compatible with the standard and previous groundings. This leads to deal with aspectual templates and their binding mechanism in a homogeneous and consistent way. In Section 7 an algorithm is formulated for the construction of a model resulting from the binding of an aspectual template to a model, following the semantics by copy informally specified in UML. This algorithm treats complete as well as partial binding of aspectual templates to (template) models. Following this formalization, reusable technology and tooling facilities offered in EMF (Eclipse Modeling Framework) are presented in Section 8. Finally, before concluding with perspectives, Section 9 discusses the generalization of the presented results inside and outside the scope of UML.

2 Background on UML templates

In this section, a synthesized reminder on UML templates is presented in order to ground the study.

2.1 The UML template construct

In the UML standard, a template is a model which exposes some of its model elements as parameters. Examples are classes or packages called class templates or package templates respectively. To specify its parameters, a template owns a signature. A template signature is a list of formal parameters where each parameter refers to an element of the template model. Templates have also a specific graphical notation which consists in superimposing a small dashed rectangle containing their signature on the top right-hand corner of the corresponding symbol.

Templates allow to define other models thanks to parameter substitution, declared in a dedicated "binding relationship". A binding relationship links a "bound model" to a template (from which it was obtained) through the specification of a set of template parameter substitutions that associate formal parameters of the template to actual elements of the bound model. Constraints of the standard only impose that the type of each actual model element must be a subtype of the corresponding formal parameter one.

The semantics of the binding relationship is specified in UML as follows: "The presence of a TemplateBinding relationship implies the same semantics as if the contents of the template owning the target template signature were copied into the bound element, substituting any elements exposed as a formal template parameter by the corresponding elements specified as actual parameters in this binding."([44], page 626). Correctness of the binding logics was formulated by OCL constraints in [8]. Note that the expansion of the template in the bound element can be made explicit graphically or not. See Figure 1 for an example of explicit expansion, other examples in the paper will use the implicit notation to suggest a constructive process.

Figure 1 shows two samples of UML templates and their binding. On the left of Figure 1(a), template class *Stack* is parameterized by *Element* of type *Class* and *Max* of type *int* which are respectively substituted by *Plate* and *15* in *PlatesStack*. The right side of Figure 1(b) is an example of a package template used here to model the well-known Observer Pattern parameterized by its *Subject* class, *Observer* class and the observed

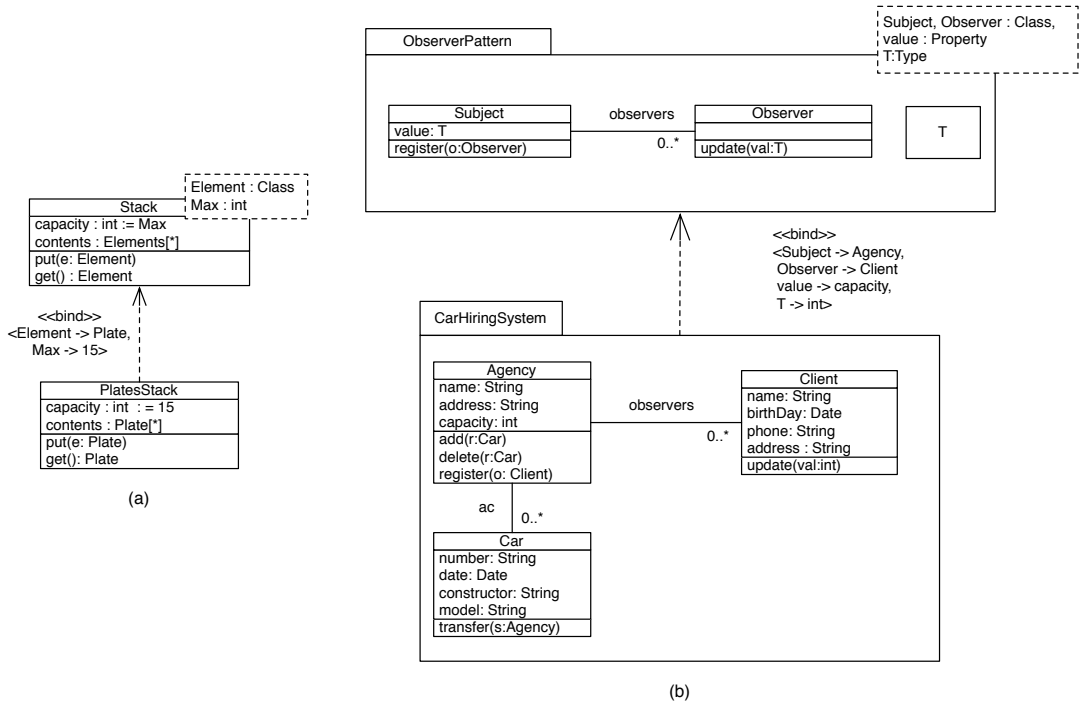


Fig. 1 Examples of a class (a) and a package (b) templates in UML

value attribute of type *T*. It is used here for the design of a "Car Hiring System" which will be used as a case study throughout the paper. This system represents cars, agencies and their clients and may offer "renting", "car search" and "stock management" functionalities among agencies. In the figure, the *Observer Pattern* template is used to install a functionality between *Agency* and *Client* for observing car availability. This design choice is specified by the binding relationship between *CarHiringSystem* and the *ObserverPattern* template with the following substitution: *Subject* to *Agency*, *Observer* to *Client*, *value* to *capacity* and its type *T* to *int*. As a result of the binding, *CarHiringSystem* includes the model structure of the Observer Pattern, after substitution was made.

Finally, a bound element may have multiple bindings, possibly to the same template. In that case it is stipulated in the standard that the bound model gets the content of each binding considered in isolation. UML allows complete and partial binding. Partial binding occurs when not all formal template parameters are substituted. For that, the UML specification only states that the unsubstituted formal template parameters are formal template parameters of the bound element ([43], page 634), which is itself a template as a consequence.

2.2 The UML template metamodel

The "Templates package" in the UML metamodel [44] introduces four main classes for their structural representation: *TemplateSignature*, *TemplateableElement*, *TemplateParameter* and *ParameterableElement* (see Figure 2). *TemplateBinding* and *TemplateParameterSubstitution* metaclasses are both used to bind templates (see Figure 3).

UML elements that are subclasses of *TemplateableElement* can be parameterized. *Classifiers*, in particular classes, and *Packages* are templateable elements[1]. The set of template parameters (*TemplateParameter*) of a template (*TemplateableElement*) are included in a signature *TemplateSignature*. A *TemplateParameter* stands for a formal template parameter and exposes an element owned by the template thanks to the *parameteredElement* role. Only parameterable elements (*ParameterableElement*) can be exposed as formal template parameters of a template or specified as actual arguments in a template binding. In particular, *Classifier*, *PackageableElement*, *Operation* or *Property* are parameterable.

The notion of template binding (*TemplateBinding*) allows to specify the use of a template for a given model (see Figure 3). A template binding is a directed relationship labeled by the << *bind* >> stereotype from the bound element (*boundElement*) to the template (*signature*). A template binding specification owns a set of template parameter substitutions (*TemplateParameterSubstitution*). A substitution associates a formal parameter of the template signature to actual parameterable elements of the bound.

Figure 4 shows an excerpt of the instantiation of this metamodel for the example described in Figure 1(b).

[1] See Section 9.1 for exhaustive lists of templateable and parameterable elements.

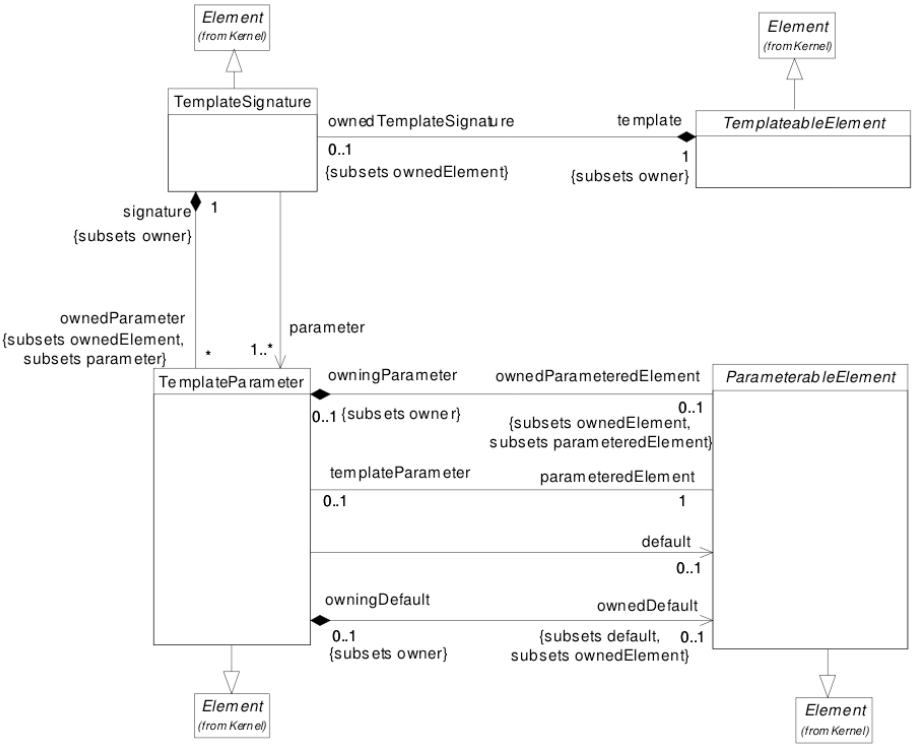


Fig. 2 UML template metamodel [43]

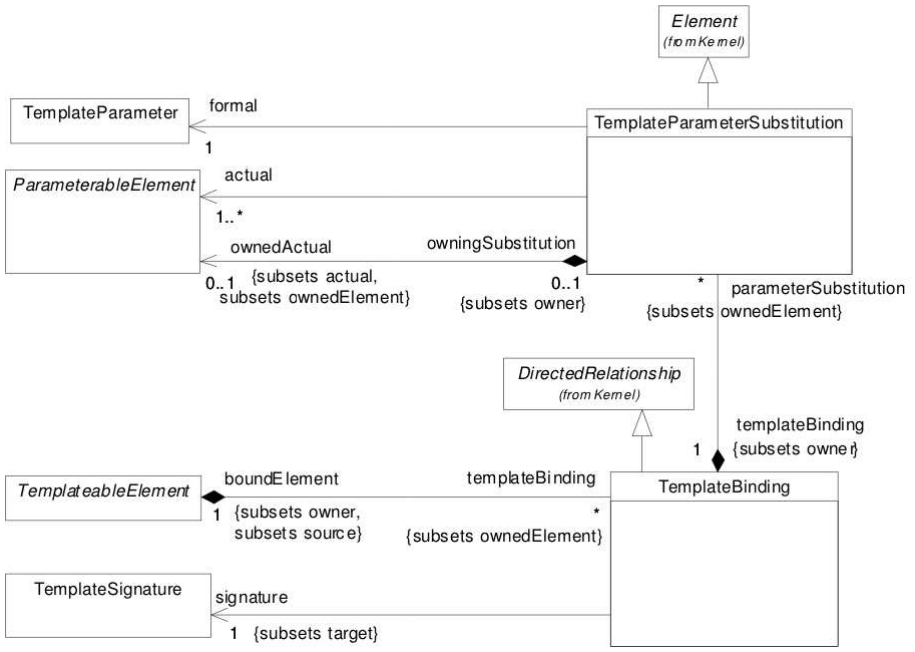


Fig. 3 UML template binding metamodel [43]

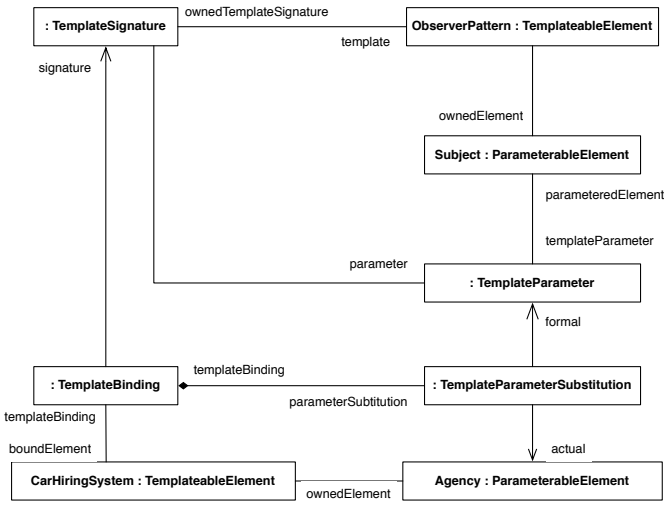


Fig. 4 Excerpt of the object diagram for the *CarHiringSystem* Package

It depicts the substitution between the *Subject* formal template parameter and the actual *Agency* argument of the bound *CarHiringSystem*.

Finally, the UML specification also introduces basic constraints for checking the definition of templates and their binding. These constraints check that:

- Elements exposed as parameters in the template signature are owned by the model being templated.
- In a substitution, the formal parameter and the corresponding actual argument have compatible metatypes.
- Each parameter substitution refers to a formal template parameter of the target template signature.
- A binding specification contains at most one parameter substitution for each formal template parameter of the target template signature.

These constraints which are general will be also valid for aspectual templates.

3 Aspectual usage of UML templates in existing works and issues

In this section we present major works which refer to UML templates for aspectual needs and identify the issues.

3.1 Existing works

The UML template construct allows to capture high-order models which represent recurrent structures. Applications of this construct are numerous and range from the modeling of generic classes to pattern formulation as exemplified previously but also aspect-oriented modeling. Few works have exploited the template construct for aspectual needs. In these works, so called "aspectual templates" aim to inject new functionalities into various base models. The capacity of such templates to expose some of their elements as parameters is exploited to specify the model structure required for making the injection possible.

The Theme approach [14] proposes means for aspect-orientation with Theme/Doc in the analysis phase and Theme/UML in the design phase. In Theme/UML, aspect models (called Themes) are specified using UML template packages containing class and sequence diagrams. Template parameters can be classes, operations or attributes. A relation (named "bind") is used to express the composition of a Theme and a base model. This relation binds the template parameters to concrete modeling elements of a base model, possibly using wildcard and multiple times.

Reddy et al. [36] describe an aspect-oriented modeling technique in which aspect models are expressed using UML template packages containing class and sequence diagrams. The approach is similar to Theme/UML but does not directly compose an aspect model (template) with a base model (called here "primary model"). Instead, a context-specific aspect model is first created by "binding" the parameters to application-specific values. It is this context-specific aspect model which is finally composed with the base model. During the composition, elements of same type and same name are merged to form a single one into the composed model. The approach also proposes "composition directives" which are intended to refine the default composition rules. They can be used to solve conflicts across aspect and base models and remove undesirable emergent properties during composition or during analysis of the composed model. This approach also provides directives to state the composition order between aspects and the primary model.

Similar to the Theme/UML approach or the composition technique proposed by Reddy et al., Kienzle et al. [27] exploit UML package templates based on class and sequence diagrams to express reusable aspect models. In this approach called RAM (Reusable Aspect Models), composing an aspect model with a base model involves binding the template parameters to base model elements, possibly with the help of pattern-matching techniques. The resulting context-specific aspect model is then composed with the base model. In this approach, some aspects may depend on the structure or behavior provided by other aspects. Such a dependency is expressed at the model level by declaring an instantiation directive with the required aspect within the dependent aspect. This directive is exploited to correctly instantiate and compose the required aspect before it can be successfully composed with a base model.

In our previous works, we also contributed to this research by studying the construction of complex systems from aspectual templates [8, 32, 33]. It appears that the construction process requires managing complex assemblies of aspectual templates with various forms of ap-

plication. For instance, there are cases where multiple aspectual templates must be applied to a same model while other cases require to apply an aspectual template to a model resulting from another application. Moreover, aspectual templates can be composed together in order to produce richer ones. This raises issues about ordering properties of applications and the equivalence of application chains. In [33] we addressed these issues in a consistent and systematic manner. This led us to formalize properties which guarantee the correctness of composition chains and their alternative ordering capacities.

3.2 Issues

All the previous works made use of UML templates for aspectual needs. However, there remains some important issues which have received little attention or have not been addressed :

- **Granularity and consistency of template parameters:** In all mentioned approaches, aspectual templates have parameters that are generic placeholders for entities expected from any base model in order to make the application possible. Regarding the set of parameters, existing approaches impose the inclusion of parameters into the set of template constituents and treat each parameter in isolation. However, considering parameters of a model template as a simple unstructured set is not sufficient for aspectual needs. Indeed, aspectual templates inject model elements into some model whose primary structure must conform to the parameters. So, to ensure the consistency of aspectual templates, there is a need to consider their parameters as a whole and impose that they form a fully structured and consistent required model. For example, an inconsistency may be: a method (resp. an association) is exposed as a parameter without its owning class (resp. some of its end classes). The Theme and RAM approaches consider parameter dependencies but only between features (attributes and methods) and their owning class. Other dependencies due to associations, type of features and parameters of operations are not considered. More generally, none of the mentioned approaches considers the idea of parameters at a coarser granularity, that is, as a plain parameter model.
- **Correctness of aspectual template binding:** Application of an aspectual template to a model is generally achieved with a dedicated binding relationship which specifies how parameters are substituted by actual model elements. Specifying such a relationship can be complex when parameters and their substitutions are numerous or have many relationships. To prevent substitution errors and enable the construction of a consistent resulting model, the correctness of this relationship should be checked automatically. Such checking must ensure that all elements from the source model conform to the types and structural constraints required by the parameters. Regarding existing works, we found that this issue is only considered in the Theme and RAM approaches but with the restriction on the granularity of parameters as indicated previously. In Theme, rules expressed in OCL are given to check the conformity between parameters and their substituted elements but these rules only concern a small set of allowed parameters. RAM also ensures a correct binding of parameters for classes and their features but the corresponding rules are only described informally and implemented in the TouchRAM tool [2], so that they are neither really reusable nor accurately comparable.
- **Aspectual templates composition:** The main interest of composing aspectual templates by themselves is to enable the construction of new and richer ones. Resulting "off-the-shelf" aspectual templates may in turn be composed or reused to enrich models. While increasing reusability, this feature has not received much attention although it raises several questions such as: what about the binding strategies between formal parameters themselves, the capacity of partial binding and the propagation of unbound parameters from the composed template to the resulting one? Answering these questions is mandatory to preserve genericity and hence reusability of model templates. Among existing works, only the RAM approach supports template composition through dependency relationships. In RAM, binding between formal parameters and propagation of unbound parameters are supported but these capacities are only offered for classes and their features without considering all parameters as a model. In addition, precise formulation of rules for the propagation of parameters and the consistency of the resulting template are not provided. More generally, and to our knowledge, the issue of composing together aspectual templates with parameter models, completely as well as partially, has not been studied.

All of these require a consistent formalization of the standard template construct semantics and its binding mechanism in case of aspectual interpretation. Indeed, despite the importance of being standard-compliant for understanding, reuse and interoperability reasons, existing works are unclear about how they interpret the standard. Works like [36] and [27] have their own modeling of template parameters and template binding. However, they do not give details of their meta-level definition. Consequently it is difficult to relate them with UML constructs and therefore get clear definition and proper semantics of aspectual templates in relation with the standard. It is the subject of the present contribution to state this semantic interpretation of the standard with the

only but strong constraint that parameters must form a model.

In the next section we present the enhancement of the UML template construct and its binding mechanism dedicated to their aspectual usage. This enhancement is fully-compliant with UML. Firstly, because formalized aspectual templates are full UML templates so that they can take place, at least, in any MDE practices where UML templates are needed. Secondly, because it respects the openness of the standard to other kinds of templates (not all UML templates have to be aspectual ones).

4 Enhancing UML templates for aspectual usage

This section presents our proposal for the definition and application of aspectual templates in the context of UML. First, we propose to enforce the template parameters to form a full parameter model whose candidate applications must conform in order to use the template. This requirement has consequences on the template application logics itself. As a result, we propose the specialization of this logics to take full advantage of the previous requirement in a homogeneous and consistent way. The capabilities offered for template composition and model assemblies are illustrated through a case study inspired from [12, 33, 46]. Next, typical scenarios of underlying engineering practices will be presented.

4.1 Parameters as a model

Aspectual templates have parameters that capture required elements from candidate models. Considering these parameters in isolation is the standard but is underspecified when one want to capture the full structure of a required model.

Figure 5 illustrates the issue[2]. This figure shows a package template for resource management functionalities related to a stock. As expected, all the parameters are model elements of the template core. But one can observe that they do not form a consistent model. Indeed, the *ref* property is exposed without its owning class whereas the latter is required to enable its mapping with a property contained in a base class. Similarly, the *in* association exposed as parameter is underspecified because one of its ends (the *Resource* class) is not declared as a parameter.

Figure 6 shows the preceding template where parameters were completed to form a full model required by the aspectual template. This required model specifies

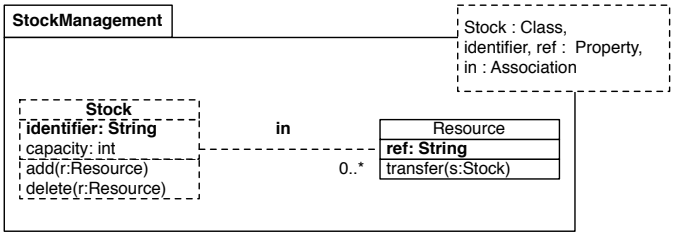


Fig. 5 Set of parameters

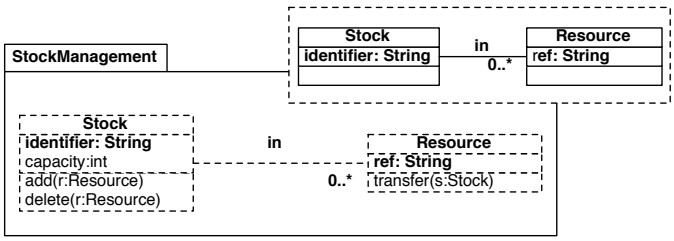


Fig. 6 Parameters as a model

the structure expected from candidate models (two connected classes with string-based attributes in the example) to correctly inject the template functionality. Graphically, this specificity is rendered by replacing the parameters list by the corresponding (parameter) model.

Other examples are provided in Figure 7 for the injection of a functionality to search resources in a stock (see *Querying*) and a counting functionality between two connected classes (see *Counting*), of which one must have a valuation method. These examples show particularly that elements of the parameter model can be either properties, operations, associations or classes. The *Allocation* aspectual template also included in the figure is particularly interesting. It gives an example where classes of the parameter model are unconnected, the purpose being to install allocation management between classes representing "Client-Product problems".

Given such kind of templates, the next section shows their application to modeling contexts as far as they conform to the required parameter model.

4.2 Binding aspectual templates

Following UML, the application of aspectual templates is supported by the *bind* relationship but it must be enhanced to take into account the "parameter as a model" requirement. This has the following consequence : a bind relationship used for aspectual template application is based on the substitution of the parameter model by a conforming sub-structure of the base model.

Consider the case study of the car hiring system and assume one wants to offer facilities for searching a specific car or client and performing car allocation. For this (see Figure 7), useful aspectual templates may be applied to its base model (center-left in the figure). Sub-

[2] In order to highlight parameter constituents in the template core, class boxes and association lines are dashed and names are bolded. Other constituents correspond to injected elements.

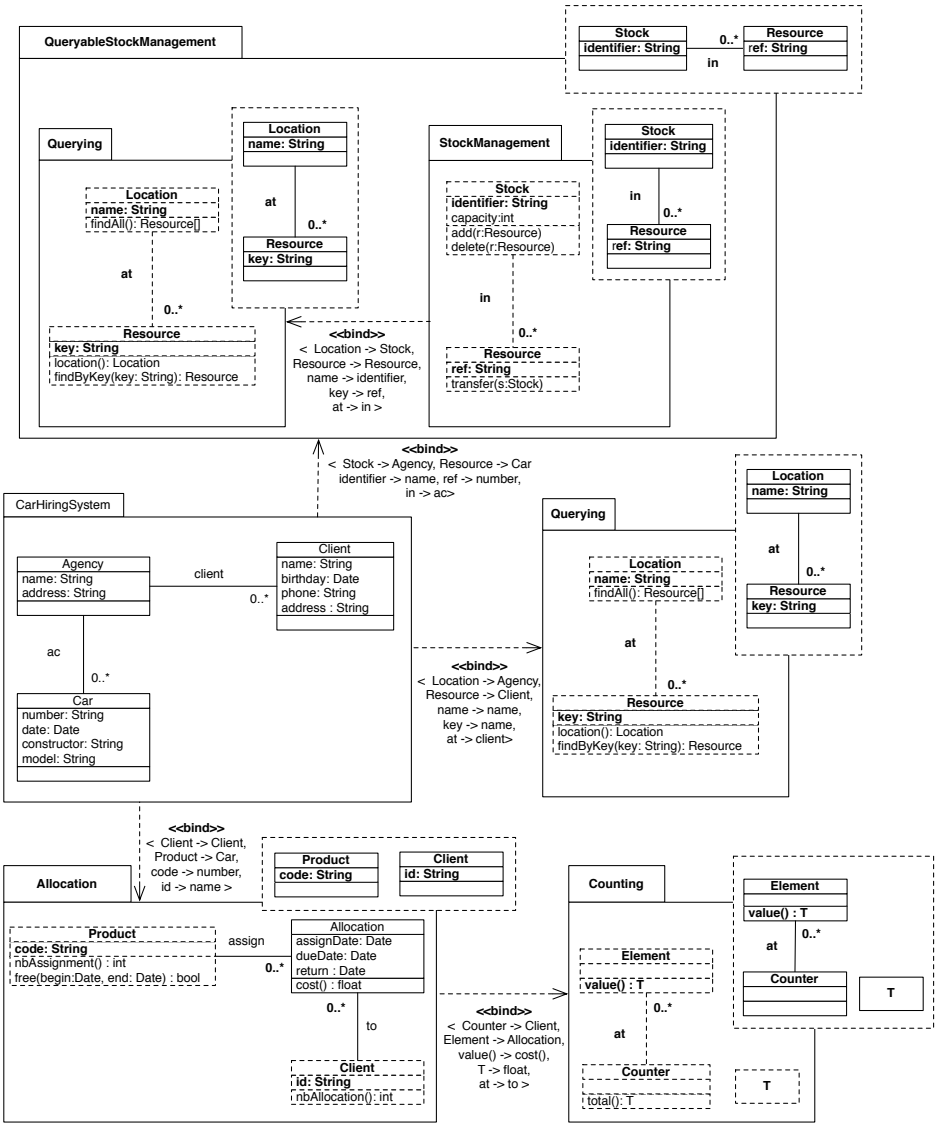


Fig. 7 Example of model assembly

sequent template application depicted in this figure will be detailed in the following. Figure 8 shows the expected result.

Compared to UML, aspectual template binding needs specific conformance: to be valid, actual arguments must form a model that structurally conforms to the parameter model of the aspectual template. This means the following: if a parameter of the template depends on another parameter (according to their modeling constraints), the same must apply to their corresponding bound elements; if two elements are connected by a link *l1* in the parameter model, their bound elements in the base model must be connected by a link bound to *l1*. These requirements will be ensured by a set of constraints detailed in Section 5.2.

For example the center part in Figure 7 shows the application of a *Querying* template to the base model. Parameters (*Location*, *Resource*, *name*, *key*, *at*) are substituted by actual elements of the base model (resp. *Agency*, *Client*, *Agency::name*, *Client::name*, *client*). One can verify that the structure formed by the parameter model is well-preserved by these actual arguments.

The preceding situation showed how an aspectual template may apply to a base model. But aspectual templates may also be composed together in order to obtain richer ones as promoted by the standard. This capacity is illustrated in the upper side of Figure 7 where the same *Querying* template is applied to the *StockManagement* one for enhancing it with querying facilities. It is worth noting that template to template application must allow the binding of formal parameters to those of the bound template. In the example the parameter *Location* is substituted by the parameter *Stock*. As a consequence, this may lead to their enrichment like any other bound element. For example the method *findAll()* of *Location* will be injected in the *Stock* parameter class

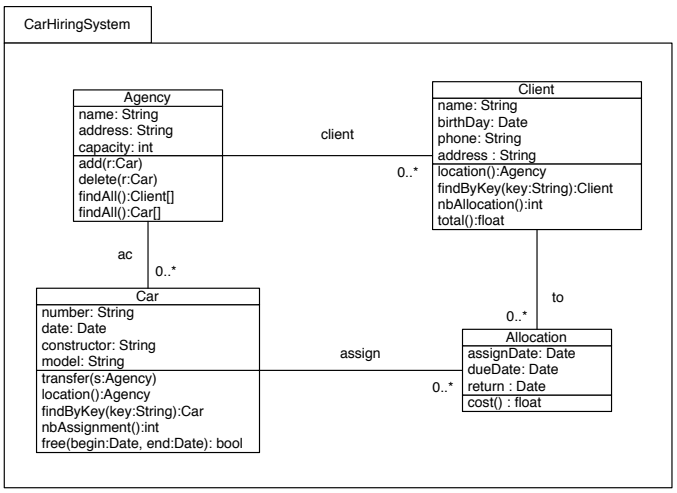


Fig. 8 Resulting system from model assembly of Figure 7

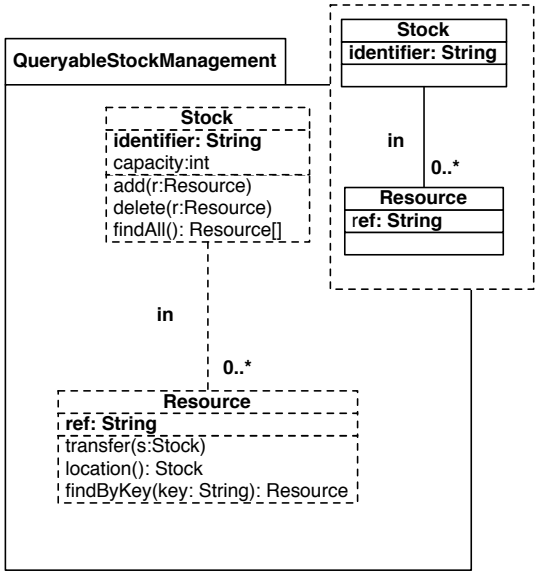


Fig. 9 Result of template to template application

with respect to specified substitions. Figure 9 shows the resulting *QueryableStockManagement* template.

At this point, it was shown how aspectual templates can be applied to base models or to other aspectual templates. Composing this primitive operation in the large leads to complex model assemblies such as the one of Figure 7. Related ordering and consistency properties must be guaranteed and were formalized in [33]. It must be possible to apply an aspectual template multiple times as it is the case for the preceding *Querying* template. Conditions are also stated to guarantee that alternative composition chains produce the same result. For example, as seen above, *Querying* applied to *StockManagement* produces the *QueryableStockManagement* template which is itself applied to the base model. An alternative would be to apply *StockManagement* to the base first then *Querying* with the same result. Another example (bottom side of the figure) is composing templates first (*Counting* to *Allocation*) then applying the resulting template to *CarHiringSystem* compared to sequencing the applications (*Allocation* to *CarHiringSystem*, then *Counting*).

This case study shows how aspectual interpretation of UML templates allows to define new systems from assemblies of prefabricated model templates with flexibility. From a much more user-centered point of view, typical engineering practices and concerns are presented in the following.

4.3 Aspectual Template Oriented Engineering in UML

Template Oriented Engineering as permitted by UML mainly involves two user roles which may be played alternatively by project contributors : designers of model templates and application modelers. When they want to use UML templates for aspectual needs, guaranteeing that parameters form a model must help them in their specific usage for building complex models from reusable ones, as in the motivating case study above (Figure 7). Figure 10 shows a typical scenario involving template designers on the left and application modelers on the right with respective activities around a model repository containing templates and models that they share.

Designers of aspectual templates are mainly concerned with "design for reuse" and the constitution of libraries ("models off the shelf"). They have to isolate the typical ("minimal") model of systems (the parameter) to which the specific reusable functionalities will apply. This is *Model Parameterization* (activity (A) in Figure 10). For example consider stock management (Figure 6), functionalities for adding/deleting/transferring resources to and from stocks may be installed in any application context which have stocks of resources, leading to the shown parameter model. Facilities must be offered to help them in their parameterization task:

- Verification that the chosen template parameters form a model.
- Automatic recommendation for the completion of the parameter model from the model elements chosen to be exposed as parameters. For example, selection of the *in* association and *ref* property as parameters requires the elicitation of ending and owning class *Resource* as parameters also. This elicitation can be automatically recommended to template designer of Figure 5 for completing its parameterization choice (Figure 6) thanks to aspectual template logics.

Application modelers are much concerned with "design by reuse" methodology (right of Figure 10). They want to exploit model templates for their application needs in a safe manner through *template binding*. Facilities must be offered to help them in this binding task:

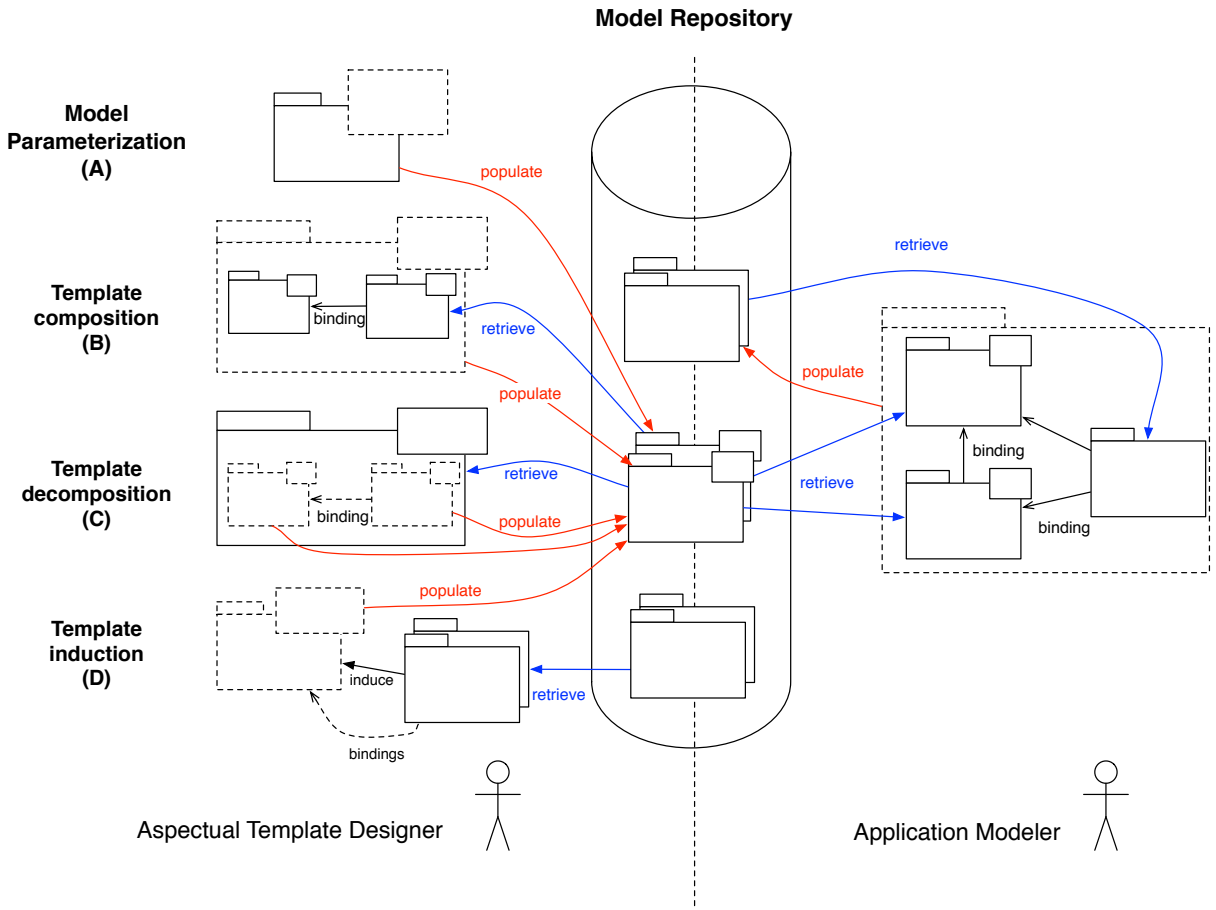


Fig. 10 Motivating engineering scenario

– Verification that binding is correct. That is, model ingredients they chose from their application context for binding to some aspectual template of the library that they want to reuse form a model which conforms to the required parameter model. For example take the context of the car hiring system and the use of the *Querying* aspectual template for its design (center part of Figure 7). Binding *Car* in place of *Client* to parameter *Resource* would be an error since association *at* bound to *client* does not link *Location* (bound to *Agency*) to *Car*.
– Automatic completion facilities similar to that offered to model template designers must also be offered but applied, this time, to the binding task. From the selection of model elements of the application context to be bound, it is possible to deduce other contextual model elements needed for the completion of the parameter model required by the template. Consider the preceding example, by selecting class *Agency* and association *at* to be bound respectively to *Location* and *at* in the model parameter, binding of class *Client* to *Resource* can be deduced. Of course it often happens that completion is multiway and aiding tools have to propose alternatives to users thanks to aspectual template logics. For example only binding *Agency* to *Location* must lead to the proposal of alternative bindings of the *at* association and the *Resource* class (with its *key* attribute of type *String*) either to the association *client* and its ending class *Client* or to the association *ac* and its ending class *Car*, with any of their *String* attribute as a parameter value.

Binding activity is iterative and compositional, allowing the construction of complex systems by successive application of model templates following rich model assemblies such as the one of Figure 7. The initial model of the system (*CarHiringSystem*, center left on the figure) is bound to aspectual templates that it conforms to. This produces enriched models of the system to which aspectual templates may apply and so on. At each step, preceding helping facilities apply again in order to verify the applicability of template bindings and/or help in completing bindings from intermediate enriched version models of the system.

Far beyond the *template binding* activity of application modelers who apply step by step aspectual templates to (nested) models within their application context, template binding does also concern designers of

aspectual templates but this time between model templates themselves in order to build richer aspectual templates from the composition of other ones. This is "Aspectual template composition" (activity (B) of template designers in Figure 10). Preceding verification and completion facilities must also apply here but with the specificity that (partial) parameter binding may apply to model elements that could be parameter elements of the bound template model.

Finally it is worth noting that, backwardly, identification of candidate models of functionalities for parameterization may come either from scratch as initially (see primary activity (A) of template designers), either:

- by decomposition of a previously identified complex template (see "Template decomposition" activity (C) in Figure 10) which leads to identify finer ones.
- by induction from previously designed models of systems which share common functionalities (see "Template induction" activity (D) in Figure 10).

These latter activities of template designers combine parameterization and binding tasks and call for verification that source models (coming from an application context or from a complex template) are binding to the identified template a posteriori.

All these engineering practices must be controlled in a homogeneous and consistent manner with the provision that aspectual interpretation of UML templates and their binding semantics are precisely and rigorously stated in order to guide users who want to exploit UML templates this way and help in ensuring the correctness of computations made by automatic processes. For this we present in the following an assertional semantics of this interpretation of the standard in OCL. The resulting formalization can be used in any situation where the concept of aspectual templates is needed such as automatic MDE engines and software design environments. It is the power of such an assertional (tool-independent) formalization to offer rules (OCL constraints) for the definition and formal specification of the standard constructs enhanced as so and more, to exploit them for interactive or automatic needs. For example, as seen above, thanks to their "verification side", rules can be used to control editing and the correctness of built models. Thanks to their "deductive side", rules can be used to offer facilities such as completion and data inference based on user (partial) entries. At a practical level such facilities will be presented in Section 8.

5 From UML templates to aspectual templates

This section presents the semantics of aspectual interpretation of UML templates and their binding using the OCL logics. As a basis, we concentrate here on "complete binding", "partial" one will be the subject of specific Section 6. Constraints apply to *TemplateSignature* and

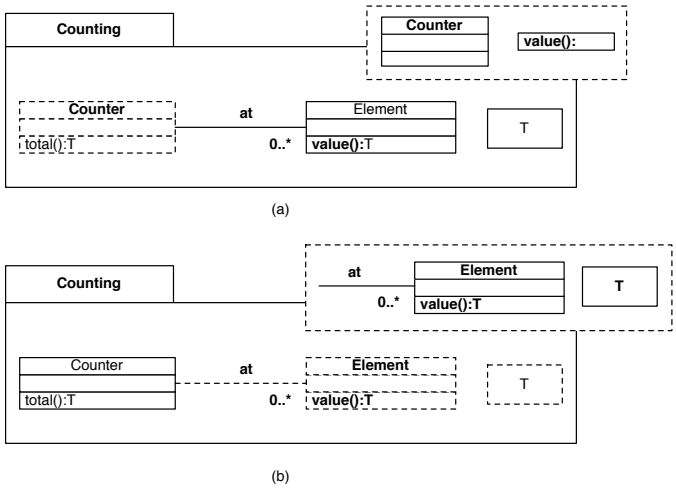


Fig. 11 Examples of not well-formed aspectual templates

TemplateBinding metaclasses. Subsection 5.1 presents constraints associated to the *TemplateSignature* metaclass for checking that parameters of an aspectual template form a valid model. Subsection 5.2 is dedicated to the correctness of aspectual template binding. For sake of simplicity, we only consider package templates consisting of classes with their features (properties and operations) linked by binary associations.

5.1 Checking template parameters

The specificity of an aspectual template compared to a general one comes from refining the semantics of the *TemplateSignature* metaclass. The signature of an UML template considers the set of parameters as individual parameters while the aspectual template signature imposes that this set forms a full well-formed model. That is the aim of the constraints formulated in this section.

Figure 11 shows some not well-formed aspectual templates which are variants of the *Counting* template (Figure 7) and will be used to explain constraints of this subsection.

Let us start with features of a class. If a feature is a parameter, its class must also be a parameter. This is ensured by the following constraint (number 1). Counterexamples are an operation without its owning class (Figure 11(a): *value()* without *Element*) or an attribute without its owning class[3].

```
-- [1] Owning classes of parameter features must
      also be parameters:
context TemplateSignature inv :
self.ownedParameter->forAll(
  param : uml::TemplateParameter |
  let pe : uml::ParameterableElement = param.
      parameteredElement in
  pe.oclIsKindOf(uml::Feature) implies
```

[3] To determine if a parameterable element is a formal template parameter, we use the standard UML query (see Templates Section in [43]): `ParameterableElement::isTemplateParameter()`

```
  let ownerClass : uml::Class
  = pe.oclAsType(uml::Feature).owner.oclAsType(uml
      ::Class) in
  ownerClass.isTemplateParameter()
)
```

Constraint 2 deals with the consistency of associations. If an association is a parameter, its ending classes must also be parameters. Figure 11(b) does not respect that constraint because *Counter* is not a parameter.

```
-- [2] Ending classes of a parameter association
     must also be parameters:
context TemplateSignature inv :
self.ownedParameter->forAll
( param : uml::TemplateParameter |
  let pe : uml::ParameterableElement = param.
      parameteredElement in
  pe.oclIsKindOf(uml::Association) implies
  let asso : uml::Association =
  pe.oclAsType(uml::Association) in
  asso.memberEnd->forAll( member | member.type.
      isTemplateParameter())
)
```

The two following constraints (3 and 4) check the typing of features that are parameters[4]. In case of a property, constraint 3 checks that its type is also a parameter. Constraint 4 is similar in case of an operation: it checks that arguments and return value types are also parameters[5]. For example in Figure 11(a), the type T of operation *value()* is not present in the signature, constraint 4 is violated. The same constraint is respected in Figure 11(b).

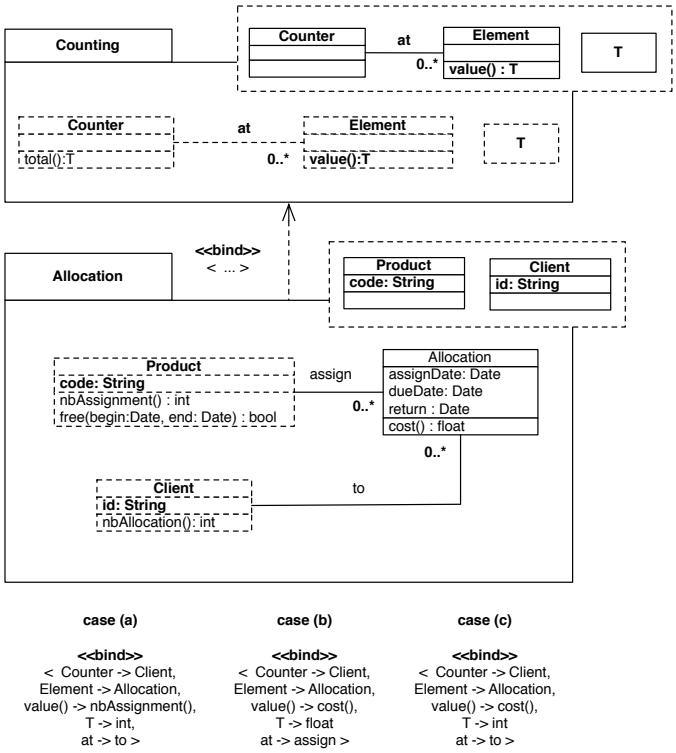
```
-- [3] The type of a parameter property must also
     be a parameter:
context TemplateSignature inv :
self.ownedParameter->forAll(
  param : uml::TemplateParameter |
  let pe : uml::ParameterableElement = param.
      parameteredElement in
  pe.oclIsKindOf (uml::Property) implies
  pe.oclAsType(uml::Property).type.
      isTemplateParameter()
)

-- [4] Types involved in a parameter operation
     must also be parameters:
context TemplateSignature inv :
self.ownedParameter->forAll(
  param : uml::TemplateParameter |
  let pe : uml::ParameterableElement = param.
      parameteredElement in
  pe.oclIsKindOf (uml::Operation) implies
  pe.oclAsType(uml::Operation).ownedParameter->
      forAll
  (p : uml::Parameter | p.type.isTemplateParameter
      ())
)
```

5.2 Checking template binding

This section presents the set of constraints related to aspectual template complete binding, that is the conformance between the formal parameter model and the actual one. Figure 12 illustrates non-conformance of three bindings between *Allocation* and *Counting* templates. The two following constraints allow the checking of model

[4] This does not apply to primitive data types.

[5] In UML, the *ownedParameter* role for an operation covers in, out and return parameters.



case (a)

<<bind>>
< Counter -> Client,
Element -> Allocation,
value() -> nbAssignment(),
T -> int,
at -> to >

case (b)

<<bind>>
< Counter -> Client,
Element -> Allocation,
value() -> cost(),
T -> float
at -> assign >

case (c)

<<bind>>
< Counter -> Client,
Element -> Allocation,
value() -> cost(),
T -> int
at -> to >

Fig. 12 Examples of conformance checking for actual parameters

structure. Constraint 5 focuses on the preservation of owned/owner relationships.

Binding (a) in Figure 12 illustrates this checking. The formal parameter *value()* is owned by the formal parameter *Element*. As a consequence, the actual parameter associated to *value()* must be owned by *Allocation* which is the substituted class of *Element*. As *nbAssignment()* is owned by the *Product* class, the constraint 5 is violated.

```
-- [5] Owning relationships between formal
     parameters must be preserved in their
     corresponding actual values:
context TemplateBinding inv :
self.parameterSubstitution->forAll (s1, s2 |
  s1.formal.parameteredElement.owner = s2.formal.
      parameteredElement
  implies s1.actual.owner = s2.actual
)
```

Constraint 6 relates to the preservation of association structures. In Figure 12(b), the member ends of the substituted association of *at*, that is *assign*, must be the substituted classes of *Element* and *Counter* parameters : *Allocation* and *Product* contrarily to given *Allocation* and *Client*. So, constraint 6 is violated.

```
-- [6] Member ends of a formal parameter
     association must be substituted by member ends
     of its actual value:
context TemplateBinding inv :
self.parameterSubstitution->forAll(s1, s2 |
  let asso:uml::ParameterableElement = s1.formal.
      parameteredElement in
  let cla:uml::ParameterableElement = s2.formal.
      parameteredElement in
  (asso.oclIsKindOf (uml::Association) and cla.
      oclIsKindOf (uml::Class)
   and asso.oclAsType(uml::Association).memberEnd
      ->collect(type)
```

```
    ->includes(cla.oclAsType(uml::Class))
    implies
     s1.actual.oclAsType(uml::Association).
          memberEnd->collect(type)
     ->includes(s2.actual.oclAsType(uml::Class))
  )
)
```

The two following constraints focus on property substitution (constraint 7) and operation substitution (constraint 8). For a property parameter, its type must be substituted by the type of the substituted property.

```
-- [7] The substituted type of a formal parameter
     property must be the type of its actual value:
context TemplateBinding inv :
self.parameterSubstitution->
select(formal.parameteredElement.oclIsTypeOf(uml::
     Property))->forAll(tps |
  let prop:uml::Property =
  tps.formal.parameteredElement.oclAsType(uml::
       Property) in
  let substitutedProp: uml::Property = tps.actual.
       oclAsType(uml::Property) in
  self.parameterSubstitution->exists(
   formal.parameteredElement=prop.type and actual=
        substitutedProp.type)
)
```

And finally constraint 8 deals with operations: Violation of this constraint is illustrated with the binding (c) in Figure 12: the return type of *value()* is T; as T is substituted by *int*, the return type of *cost()* is not compatible: it must be *float* instead of *int*. As a result, the *value()* operation can not be substituted by *cost()*.

```
-- [8] The substituted types of operation
     parameters must be
substituted by the corresponding types of the
     actual operation:
context TemplateBinding inv :
self.parameterSubstitution->select(formal.
     parameteredElement.
oclIsTypeOf(uml::Operation))->forAll (tps |
let formalOp : uml::Operation =
  tps.formal.parameteredElement.oclAsType(uml::
       Operation) in
  let actualOp : uml::Operation =
  tps.actual.oclAsType(uml::Operation)  in
  formalOp.ownedParameter->size()=actualOp.
       ownedParameter->size() and
  Sequence{1..formalOp.ownedParameter->size()}->
       forAll ( index |
    let memberFormalOp : uml::Parameter =
    formalOp.ownedParameter->asOrderedSet()
    ->at(index).oclAsType(uml::Parameter) in
    let memberActualOp : uml::Parameter =
    actualOp.ownedParameter->asOrderedSet()
    ->at(index).oclAsType(uml::Parameter) in
    self.parameterSubstitution->exists(
      formal.parameteredElement=memberFormalOp.
           type and
      actual=memberActualOp.type )
  )
)
```

p

6 Partial binding of aspectual templates

In the previous section, we have presented basics of the aspectual enhancement of UML templates in the general case of "complete binding" (all parameters being substituted). In this section, we concentrate on "partial binding". Partial binding occurs when only a subset of the parameters are substituted so that unbound ones remain parameters in the resulting model which is therefore a template. This feature aims to define new templates with richer parameter models, resulting in a more powerful and flexible composition logics between templates. This facility allows to hierarchically compose templates and does increase their reusability.

In the following, we first explain partial binding in the context of aspectual templates. This requires additional rules for ensuring the consistency of partial binding, particularly the well-formedness of the parameter model in the resulting template. Then we present these rules and give their OCL formulation.

6.1 Partial binding in detail

To get a glimpse of partial binding, let us consider an aspectual template capturing the *Observer* pattern and its application to the *StockManagement* template introduced in Section 4.1. The aim of this application is to get a new template expressing the functionality for observable stock management of resources. Figure 13 shows this application. It consists in the following substitutions of the *Observer* template parameters: the *Subject* class is bound to the *Stock* class, the *value* attribute is bound to the *capacity* attribute and its type T is bound to the *int* datatype. As we can notice, this application of the *Observer* template is partial: *Observer* class and *observers* association included in the parameter model are not substituted by elements of the bound template. In this application, none of the classes contained in the *StockManagement* template is intended to play the *Observer* functionality. Regarding this example, it is worthwhile to note that, without partial binding, it would not be possible to compose the involved templates.

In case of partial binding, how unbound parameters are handled in the application process needs to be determined. Several strategies are possible for these parameters: simply ignore them; include them in the core of the resulting template while dropping their status as parameters; propagate them in the parameter model of the resulting template. The last strategy is the one specified in UML ([43] page 634): "In case of partial binding, the unbound formal template parameters are formal template parameters of the bound element.". Compared to other ones, this strategy offers two main advantages. First, it respects the high-order status of template parameters compared to other model elements, which is to abstract elements expected from any candidate model to fulfill the template functionality. Other strategies break this principle. Second, this strategy allows to obtain new templates with enriched parameters. This strategy of the standard is fully respected here. However, as explained in the following, it must be made consistent with aspectual templates so that parameters must form and remain a full model during binding.

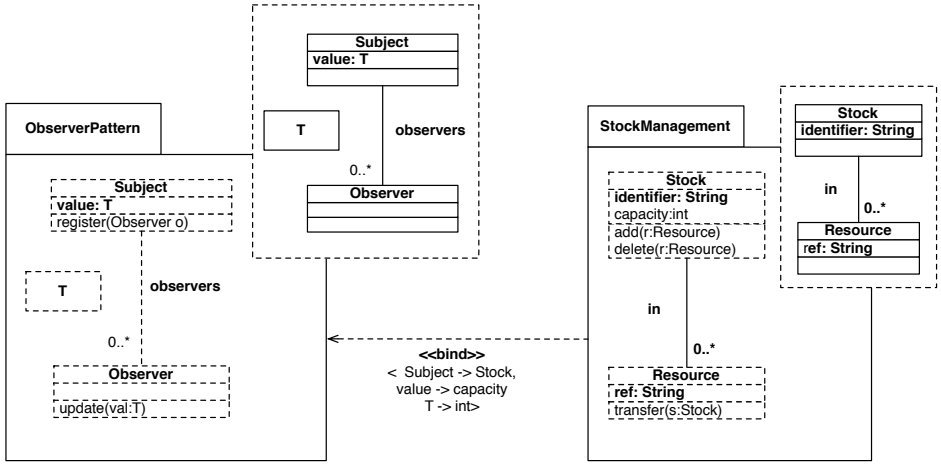


Fig. 13 Example of unsubstituted parameters: classes and associations

Right of Figure 14 illustrates this strategy by means of the *ObservableStockManagement* template resulting from the previous partial binding (Figure 13). One can observe the two following points. First, the unsubstituted *Observer* class and *observers* relationship have been inserted into the template core but keeping their parameter status so that they are injected in the parameter model of *ObservableStockManagement*. The insertion of these unsubstituted parameters is achieved with respect to specified substitutions, causing the adaptation of their owning elements. See for example the substitution of *T* by *int* in the *update* method. Second, both elements are also parameters of the *ObservableStockManagement* template, resulting in a richer parameter model. These added parameters should be substituted in further applications of the obtained template. More generally, following the standard strategy, the parameter set of the new template is determined by the union of the bound model parameters and the source unsubstituted model parameters. To follow aspectual template semantics, it is essential that this augmented parameter set forms a consistent model. The next subsection studies this issue.

The previous example of partial binding illustrates the case of unsubstituted class and association parameters. Owned elements like attribute and operation parameters can also be unsubstituted, even if their owning class parameter is bound. As an illustrative example of this capacity, let us modify the *Querying* template used in Section 4.2 a bit. The modification consists in adding an *address* and a *date* attributes as parameters of the *Querying* template so that the functionality of searching a resource by date or location could be customized with corresponding attributes from the source domain. In addition to this modification, we also consider partial binding of this template to the previously obtained *ObservableStockManagement* template for providing observable stock management with a querying facility.

Figure 14 shows the modified parameter model of the *Querying* template and its partial application to *ObservableStockManagement*. The resulting *QueryableObservableStockManagement* template is shown in the left part of Figure 15. In this example, *address* and *date* parameters being unsubstituted, they are included in the parameter model of the resulting template. Indeed, these attributes are still needed to fulfill the functionality added to the resulting template for searching a resource by date or location. Regarding the need for consistency of the resulting parameter model, unsubstituted attributes and operations parameters require that their owning class in the bound be a parameter (see *Location* substituted by *Stock* in Figure 14). This requirement is necessary to ensure that such parameters have an owning class in the resulting parameter model.

From the successive partial applications described previously, we finally obtain a quite rich aspectual template. This template combines several functionalities for observing, searching and managing resources at the same time. This template can be capitalized as a value-added reusable model in repositories and be further applied to construct systems. Figure 15 illustrates the application of this template to the *CarHiringSystem* so that agencies can manage their stock of cars and clients can observe the availabilities.

Figure 16 shows the overall resulting model. It is the result of the partial application chain as schematized on the left in Figure 17. It is useful to highlight that the same result model could be obtained through alternative ordering composition processes of the involved templates such as exemplified on the right of Figure 17: first applying *Querying* and *StockManagement* (in any order) and then *ObserverPattern*. This equality emphasizes the compatibility between partial and complete binding, therefore the consistency of the partial binding strategy. During system design, partial and complete binding can be mixed in an effective way thanks to ordering properties, some parts originating from templates designed from scratch, other parts coming from templates obtained by valuable composition of other templates.

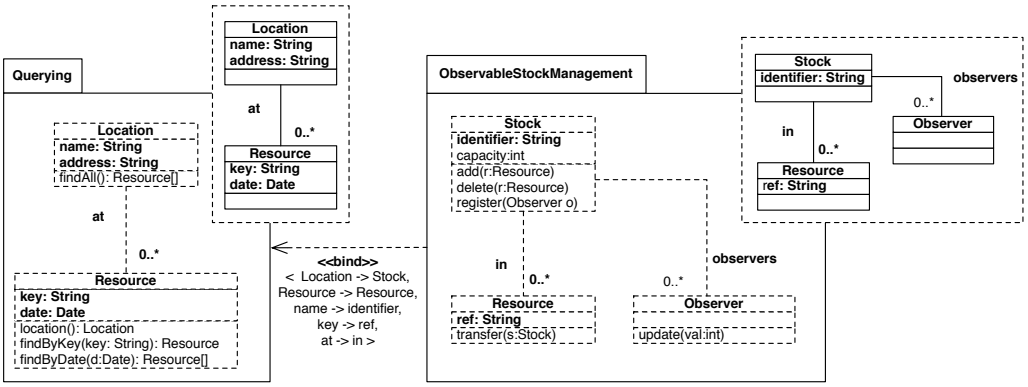


Fig. 14 Example of unsubstituted parameters: attributes

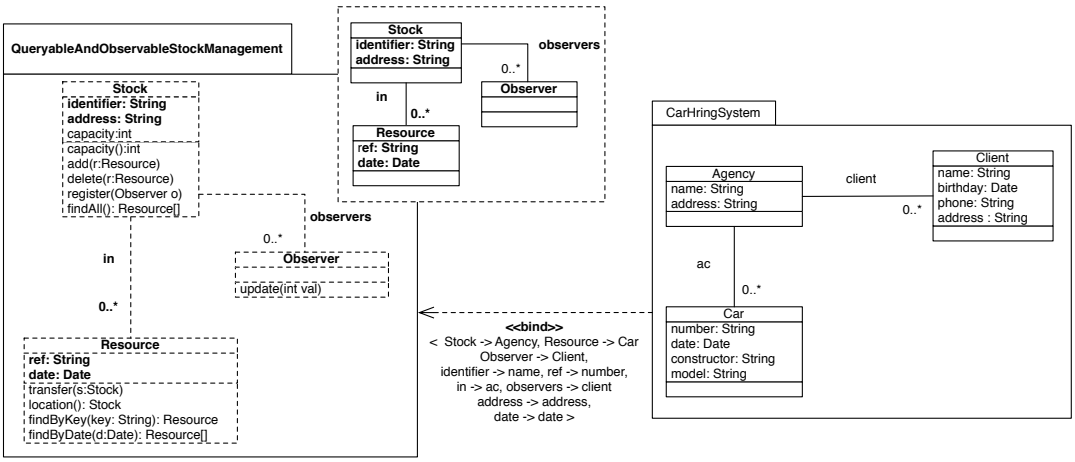


Fig. 15 Binding QueryingObserverStockManagement template to the base model

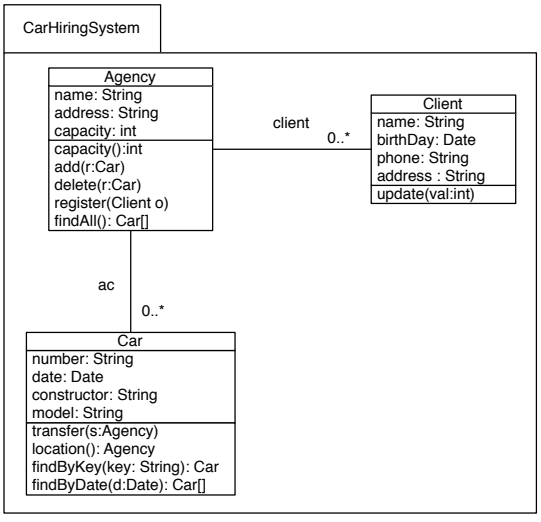


Fig. 16 Model Resulting from QueryingObserverStockManagement Application

6.2 Checking partial binding

The constraints described in Section 5 for the consistency of complete template binding remain valid in case of partial one. But partial binding also calls for specific rules to ensure its consistency. These rules are the following:

1. The subset of substituted parameters must form a well-formed part of the parameter model[6]. Intuitively in Figure 14, consider the *key* attribute parameter and its *Resource* owning parameter class which are both substituted. Substitution of this attribute implies substitution of its class. Otherwise it will not be possible to specify the class owning the substituted attribute in the bound template and as a consequence the core of the resulting template will not be well-formed. Additionally, in Figure 18(b), substitution of *observers* association without substituting the *Observer* class will yield the same kind of problem.
2. The bound model of a partial binding must be an aspectual template. This requirement is necessary to ensure that unsubstituted parameters have a context, that is the parameter model, in the resulting template.

[6] If all parameters are substituted, this property is implicitly guaranteed because they form a model by definition.

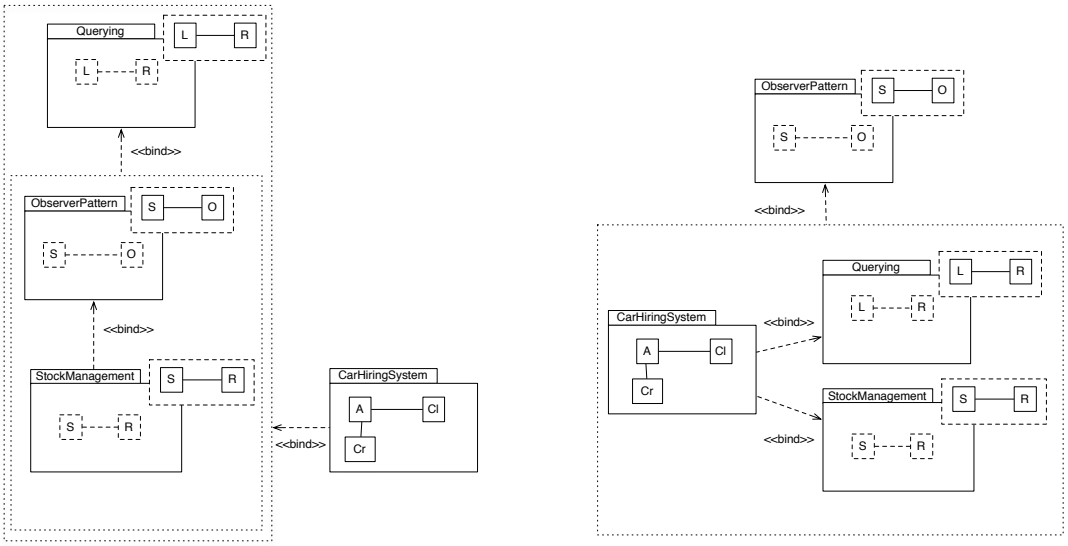


Fig. 17 Two composition processes of templates yielding the same result

3. Unsubstituted parameters must have their container or their dependent elements in the parameter model bound with elements having the status of parameter in the bound template (see the unsubstituted *date* attribute of the *Resource* class in Figure 14). This requirement is necessary to ensure that the parameter model augmented with unsubstituted parameters forms a valid model. In Figure 13, the unsubstituted *observers* association parameter is an example satisfying this requirement for its ending *Subject* and *Observer* parameter classes : the first one is bound to the *Stock* class parameter and the second one is unsubstituted, so that it remains a parameter. In Figure 14, one can see that the requirement is also fulfilled since unsubstituted *date* and *address* parameter attributes have their respective classes bound to class parameters in the bound template. Otherwise, such parameters would not have an owning class and the resulting parameter model would not be well-formed.

In the following, we detail these rules separately and formulate the corresponding OCL constraints. Constraints (9-12) deal with the first rule. These constraints are similar to those used for checking that parameters included in a template signature (see constraints 1-4) form a model. Here, these constraints check this property for the substituted parameters specified by the template binding. Constraint 9 checks for contained elements (features without their owning class), constraint 10 for associations (associations without their ending classes), and constraints 11-12 for features (features without their respective types).

We use Figure 18 to illustrate violation of these constraints. In case (a) of this figure, the class *Subject* which owns the substituted attribute parameter *value* is not substituted, so violates the constraint 9. Meanwhile, the

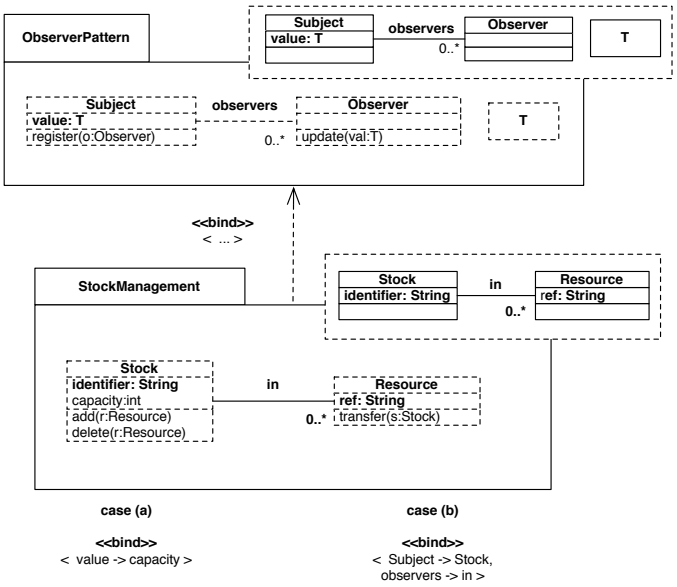


Fig. 18 Examples of formal parameters checking for partial binding

type of this attribute being parameter (T), it must also be substituted to respect the constraint 11.

In Figure 18(b), the parameter association *observers* between *Subject* and *Observer* is substituted but it has only one substituted ending class (*Subject* by *Stock*), leading to violation of constraint 10.

```
-- [9] Owning class of a substituted feature must
     also be substituted:
context TemplateBinding inv :
self.parameterSubstitution->collect(formal.
    parameteredElement)
->includesAll(self.parameterSubstitution->select(
  formal.parameteredElement.oclIsKindOf(uml::
      Feature))->collect(
  formal.parameteredElement.oclAsType(uml::Feature
      ).
  featuringClassifier)
```

```
)

-- [10] Ending classes of a substituted
     association must also be substituted:
context TemplateBinding inv :
let setpe : Set(uml::ParameterableElement) =
self.parameterSubstitution->collect(formal.
     parameteredElement)->asSet() in
setpe->forAll(pe : uml::ParameterableElement |
  pe.oclIsKindOf(uml::Association) implies
  pe.oclAsType(uml::Association).memberEnd->forAll
       ( m |
    setpe->includes(m.type)
  )
)

-- [11] The type of a substituted property must
     also be substituted:
context TemplateBinding inv :
self.parameterSubstitution->select(formal.
     parameteredElement.oclIsKindOf(uml::Property)
     )
->forAll( ps |
  let propertyType : uml::Classifier =
  ps.formal.parameteredElement.oclAsType(uml::
       Property)
  .type.oclAsType(uml::Classifier) in
  self.parameterSubstitution->collect(formal.
       parameteredElement)
  ->includes(propertyType.oclAsType(uml::
       ParameterableElement))
)

-- [12] Types involved in a substituted operation
     must also be substituted:
context TemplateBinding inv :
self.parameterSubstitution->forAll(ps : uml::
     TemplateParameterSubstitution |
  ps.formal.parameteredElement.oclIsKindOf(uml::
       Operation) implies
  let op : uml::Operation =
    ps.formal.parameteredElement.oclAsType(uml::
         Operation)
  in op.ownedParameter->forAll (p : uml::Parameter
       |
    self.parameterSubstitution->collect(formal.
         parameteredElement)
    ->includes(p.type)
  )
)
```

The second rule is ensured by the constraints 1-4. Remember that these constraints check that the signature of the bound element forms a model.

Constraints (13-16) and Figure 19 focus on the last rule. Constraint 13 checks for unsubstituted features. If a class C is substituted and one of its features is not then the class substituted for C must be parameter of the bound template. Let us illustrate this with the binding of *Counting* to *Allocation* in Figure 19 where the *Element* class is substituted by *Allocation*. As the operation *value()* is not substituted and propagated as a parameter, its class must also be a parameter (w.r.t constraint 1). Here, the binding violates constraint 13.

```
-- [13] If a parameter class c is substituted by a
     class c' and one parameter feature of c is
     not substituted then c' must be a parameter of
     the bound template:
context TemplateBinding inv :
self.parameterSubstitution->
select(formal.parameteredElement.oclIsKindOf(uml::
     Class))
->forAll ( ps : uml::TemplateParameterSubstitution
     |
  self.parameterSubstitution->collect(formal)->
       includesAll(
    self.signature.ownedParameter->
    select(parameteredElement=ps.formal.
         parameteredElement)
  ) or ps.actual.isTemplateParameter()
)
```

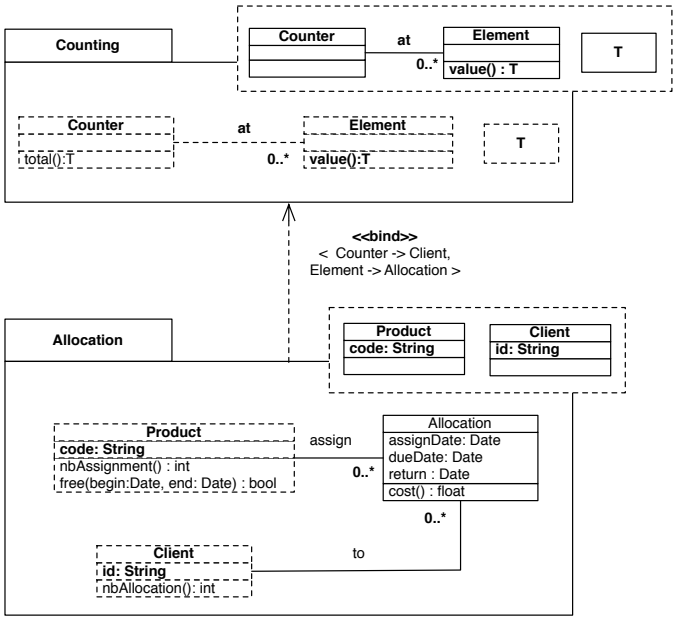


Fig. 19 Examples of checking for unsubstituted parameters

Let us refine the study of unsubstituted features. If the unsubstituted feature is a property and its type is substituted, the substituted type must be parameter of the bound template. Constraints 14 and 15 check this rule respectively for properties and operations. For example, consider the binding in Figure 13 with a substitution of T but not of *value*. Then, *value* would be propagated as a parameter in the resulting template but without its type (the result would violate constraint 3).

```
-- [14] If a parameter class c is substituted by a
     class c' and c is the type of a parameter
     property which is not substituted then c' must
     be a parameter of the bound template:
context TemplateBinding inv :
self.parameterSubstitution->
select(formal.parameteredElement.oclIsKindOf(uml::
     Class))
->forAll ( ps : uml::TemplateParameterSubstitution
     |
  self.parameterSubstitution->collect(formal)->
       includesAll(
    self.signature.ownedParameter->
    select(parameteredElement.oclIsKindOf(uml::
         Property)
      and
      parameteredElement.oclAsType(uml::Property).
           type=
      ps.formal.parameteredElement)
  ) or ps.actual.isTemplateParameter()
)

-- [15] If a parameter class c is substituted by a
     class c' and c is one of a parameter type of
     a parameter operation which is not substituted
     then c' must be a parameter of the bound
     template:
context TemplateBinding inv :
self.parameterSubstitution->select(formal.
     parameteredElement.oclIsKindOf(uml::Class))
->forAll ( ps : uml::TemplateParameterSubstitution
     |
```

```
  self.parameterSubstitution->collect(formal)->
      includesAll(
    self.signature.ownedParameter->select(
      parameteredElement.oclIsTypeOf(uml::
          Operation) and
      parameteredElement.oclAsType(uml::Operation)
          .ownedParameter
      ->exists(type=ps.formal.parameteredElement)
    )
  ) or ps.actual.isTemplateParameter()
)
```

The last constraint (constraint 16) focuses on unsubstituted associations. If a member end of an unsubstituted association is substituted then it must be parameter of the bound template. In Figure 19, constraint 16 is violated by the unsubstituted association *at* because its *Element* member end is substituted by the *Allocation* class which is not a parameter of the bound template. This would yield a propagated but dangling association in the resulting parameter model (constraint 2 is violated).

```
-- [16] If a parameter class c is substituted by a
     class c' and c is one of a member end of an
     parameter association of c which is not
     substituted then c' must a parameter of the
     bound template:
context TemplateBinding inv :
self.signature.ownedParameter->
select(parameteredElement.oclIsTypeOf(uml::
    Association))->forAll(
  p : uml::TemplateParameter |
  let asso : uml::Association =
  p.parameteredElement.oclAsType(uml::Association)
      in
  not(self.parameterSubstitution->collect(formal)
      ->includes(p)) implies
  self.parameterSubstitution->forAll( ps |
    asso.memberEnd->forAll ( propertyEnd |
      propertyEnd.type=ps.formal.
          parameteredElement
      implies ps.actual.isTemplateParameter()
    )
  )
)
```

7 Binding algorithm

This section presents an algorithm for the construction of a consistent model resulting from the binding of an aspectual template to a model or another aspectual template. It follows the semantics by copy and substitution informally specified in UML. This algorithm supports complete and partial binding.

The algorithm (see Algorithm 1) takes one input parameter which is a *TemplateBinding* specifying the target aspectual template, the bound package and a set of substitutions. The effect of the algorithm is to modify the bound package with additional elements of the aspectual template after parameter substitution.

We make the assumption that the *TemplateBinding* parameter is a valid aspectual template binding with respect to the set of rules presented in the previous sections. So, the checking of *TemplateBinding* is not included in the algorithm. In particular, when partial binding, the bound template must be an aspectual one. We also assume the following:

- The *ParameterableElement* metaclass owns an *isTemplateParameter(): boolean* operation which returns true if the element is exposed as a template parameter, false otherwise.
- The *Package* metaclass owns an *addParameter(pe:ParameterableElement)* operation which adds *pe* contained in the package as a template parameter. This operation creates and attaches a new *TemplateParameter* referencing the provided element to the *TemplateSignature* of the package.
- The *TemplateBinding* metaclass owns an *isSubstituted(pe:ParameterableElement) : boolean* query operation which returns true if *pe* is a formal template parameter bound to an actual element in the bound package, false otherwise.
- The *TemplateBinding* metaclass owns an *getActual(pe:ParameterableElement) : ParameterableElement* query operation which returns the actual argument corresponding to the *pe* formal parameter.
- *Map* is a conventional class for mapping keys to values. It is used here to memorize substitution of classes done in the first step.
- A *clone()* operation is available on parameterable elements. It creates an element of the same metatype and copies its meta-attributes.

The algorithm is based on three steps :

1. **Copy template classes into the bound package:** In this step, iteration is made over the set of classes contained in the aspectual template. If a class is not a parameter or is an unsubstituted parameter, a clone without features is created and added to the bound package. In case of an unsubstituted template class, a corresponding parameter is added to the bound package signature. Finally, in preparation to the next step, mapping is made between each template class and its corresponding bound class which is either one specified by the template binding or one created previously.
2. **Extend bound classes with features issued from the template:** This step consists in extending all the bound classes mapped to template classes with clones of their properties and operations which are not parameters or are unsubstituted ones. The algorithm iterates over the mapping set up during the first step to determine each bound class. For each added property and operation, the template classes referenced by their types are replaced by the corresponding mapped classes. In addition, unsubstituted features from the aspectual template are propagated as parameters in the bound template signature.
3. **Copy template associations into the bound package:** In this last step, the algorithm inserts a copy of template associations which are not parameters or are unsubstituted parameters into the bound package. New associations have their owned ends adapted to take into account substituted or cloned template

classes. Unsubstituted associations are propagated as parameters in the bound template signature.

The algorithm produces a resulting (template) model which is consistent with the standard bind relationship. As explained in Section 2.1, this implies that the content of the bound model includes the content of the template with any element exposed as a formal parameter substituted by the actual element specified in the binding.

8 Aspectual template technology

In the preceding sections, a formalization of aspectual templates in OCL was stated. This formalization aims at capturing common groundings when using UML templates for aspectual needs. Its ambition is to be independent of (so reusable in) any particular tool, user interaction facility, or specific automatic process. This opens the way to many possibilities for applying and implementing aspectual templates as specified in the present work. In this section, we present core technology that has been made available in the Eclipse Modeling Framework environment. Then, we present how it can be integrated into specific CASE tools through an example.

8.1 Core functionalities

Following the plugin-based style promoted by Eclipse, we have developed new plugins that offer core functionalities dedicated to aspectual templates[7]. These plugins, which rely on official EMF, UML and OCL plugins for their implementation, are the following:

- A UML profile plugin that allows to optionally apply the aspectual interpretation and its constraints. This profile is compliant with the official UML plugin provided by Eclipse. As a result, the stereotypes can be applied to any UML model using either the programming interface or any profile-compliant UML CASE tool. The profile consists of three dedicated stereotypes (Figure 20) which provide contexts for applying the specific OCL constraints. These stereotypes are :
  - *AspectualTemplate* and *AspectualTemplateSignature* related to package templates and their signature;
  - *AspectualTemplateBinding* which enhances the binding rules.

  Figure 20 shows the extension using the standard profile notation (see Profiles Section in [43]). It uses the standard "extension" association between a stereotype definition and the extended metaclass. The stereotypes must be applied conjointly to be consistent.

[7] The core functionalities and the CASE tool are available at `http://www.cristal.univ-lille.fr/caramel/aspectualtemplates`

Algorithm 1: apply(*binding* : *TemplateBinding*)

```
Data: template, base : Package, map : Map < Class, Class >
begin
    map ⟵ {};
    template ⟵ binding.signature.template;
    base ⟵ binding.boundElement;
    // Step 1: Copy template classes
    for tcl ∈ template.classes do
        if ¬tcl.isTemplateParameter() ∨
          ¬binding.isSubstituted(tcl) then
            bcl ⟵ tcl.clone();
            base.classes ⟵ base.classes + bcl;
            if tcl.isTemplateParameter() then
                base.addParameter(bcl);
            end
        else
            bcl ⟵ binding.getActual(tcl);
        end
        map ⟵ map+ < tcl, bcl >;
    end
    // Step 2: Extend bound classes
    for tcl ∈ map.keys() do
        bcl ⟵ map.get(tcl);
        // Properties
        for tprop ∈ tcl.ownedAttribute do
            if ¬tprop.isTemplateParameter() ∨
              ¬binding.isSubstituted(tprop) then
                bprop ⟵ tprop.clone();
                bprop.type ⟵ map.get(tprop.type);
                bcl.ownedAttribute ⟵
                  bcl.ownedAttribute + bprop;
                if tprop.isTemplateParameter() then
                    base.addParameter(bprop);
                end
            end
        end
        // Operations
        for top ∈ tcl.ownedOperation do
            if ¬top.isTemplateParameter() ∨
              ¬binding.isSubstituted(top) then
                bop ⟵ top.clone();
                bop.type ⟵ map.get(bop.type);
                bcl.ownedOperation ⟵
                  bcl.ownedOperation + bop;
                for tparam ∈ top.ownedParameter do
                    bparam ⟵ tparam.clone();
                    bparam.type ⟵
                      map.get(tparam.type);
                    bop.ownedParameter ⟵
                      bop.ownedParameter + bparam;
                end
                if top.isTemplateParameter() then
                    base.addParameter(bop);
                end
            end
        end
    end
    // Step 3: Copy template associations
    for tassoc ∈ template.associations do
        if ¬tassoc.isTemplateParameter() ∨
          ¬binding.isSubstituted(tassoc) then
            bassoc ⟵ tassoc.clone();
            base.associations ⟵
              base.associations + bassoc;
            for tend ∈ tassoc.memberdEnd do
                bend ⟵ tend.clone();
                bend.type ⟵ map.get(tend.type);
                bassoc.memberEnd ⟵
                  bassoc.memberEnd + bend;
            end
            if tassoc.isTemplateParameter() then
                base.addParameter(bassoc);
            end
        end
    end
end
```

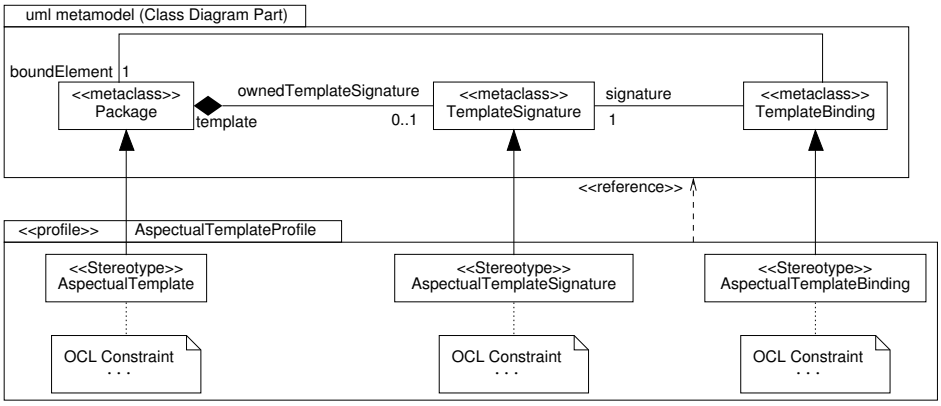


Fig. 20 UML Profile for aspectual templates

– An engine plugin that checks all specified OCL constraints. Constraints are parsed and executed using the Java API provided by the official OCL plugin. This plugin also implements the binding algorithm presented in Section 7. The algorithm implementation exploits the API provided by the UML plugin to represent and manipulate UML models in Java.
– An helper plugin that offers general facilities for querying, modifying or completing aspectual template signatures and bindings in relation with the "parameters as a model" requirement. Among the provided facilities, one allows to determine the missing parameters in aspectual template signatures and bindings. Another facility is the automatic completion of signatures and bindings to guarantee that their formal parameters form a model. Inference of parameter substitution for aspectual template bindings is also offered (see next subsection for a detailed explanation of this powerful functionality). This plugin can be helpful to develop new modeling tasks targeting aspectual templates and also to provide user assistance during aspectual template specification and binding like the automatic fixing of errors detected by OCL constraints.

All the preceding core functionalities are available to other plugins and can be easily integrated into UML tools which are compliant with the Eclipse architecture and its EMF framework. Kinds of tools that can profit from such plugins are for example UML model verifiers, transformation and construction engines as well as model editors as illustrated in the subsection 8.3.

8.2 Binding inference

Binding inference is the process of finding automatically valid parameter substitutions for an aspectual template binding specification. This feature is currently provided by the helper plugin. To provide this feature, we use a well-known algorithm for subgraph isomorphism detection proposed by Ullman in [42]. This algorithm, which is based on the principle of backtracking in combination with a forward-checking technique, enables to find all the mappings between two graphs that respect some iso-structural conditions.

In the current work, this algorithm has been adapted to find all the valid substitutions between the model parameter and the bound model with regard to the structural constraints imposed by aspectual template semantics. The adapted algorithm starts with a single mapping of a parameter to a compatible element and then gradually extends this mapping with additional ones such that the set of determined mappings always denotes a valid binding. If, at some point, the set of computed mappings does not represent a valid binding, then the process backtracks. That is, a previously mapped parameter is reassigned to another candidate element from the bound model and the conditions for the binding correctness are again tested. At each level of the execution, the forward-checking procedure is used to test whether there exists at least one mapping for each future parameter onto some element such that the conditions for valid binding hold true. This allows to avoid computation steps in the intermediate level before discovering that a parameter can not be correctly mapped.

Binding inference works in case of partial binding. The algorithm handles this case by limiting the enumeration of possible mappings to the subset of formal parameters. Furthermore, the binding inference also operates if substitutions exist for some parameters at the initial stage. In that case, the algorithm takes the given substitutions as being fixed mapping and systematically includes them in all resulting valid bindings (see the following for an example).

The capacities of the algorithm are illustrated in Figure 21 for the complete application of StockManagement aspectual template to the Car Hiring System model (see Figure 7). In this example, the binding only specifies the substitution of *Stock* by *Agency*. Other parameters are not substituted (their actuals are noted *undefined* in the figure). From this binding, the algorithm produces a set of twenty possible bindings shown in the right part

of the figure. These bindings are the only possible ones with respect to the structural constraint imposed by the parameter model with the given substitution. As we can observe, the algorithm inferred that candidate elements conforming to the structure formed by *Stock* substituted by *Agency*, *Resource* and the *in* association are the subset formed by *Agency*, *Client*, *client* or the one formed by *Agency*, *Car* and *ac*. Other substitutions found by the algorithm correspond to the substitution of the *ref* parameter of *Resource* and the *identifier* parameter of *Stock* by one of the type-compatible attributes owned by the candidate classes found for their substitution: for instance, *number*, *constructor* and *model* attributes for the *ref* parameters when *Resource* is substituted by *Car*. These results show the enumerative nature of the algorithm to find all valid substitutions.

At the user level, binding inference can be exploited to assist the modeler with the task of substituting parameters as seen in Subsection 4.3. For example, the modeler can perform inference to discover all the valid bindings and select the most convenient ones with regard to his modeling problem. From these bindings, it is also possible to compute their intersection and group them according to their common substitutions in order to reduce the set of bindings proposed to the modeler. Another way of using this facility is the completion of a binding with additional inferred substitutions. For example, a modeler can specify the binding of an attribute parameter to an existing one in the bound model and rely on the inference mechanism to determine the substitution of its class parameter by the corresponding class. A last interesting usage of binding inference is the filtering of elements that can be candidates for a particular substitution. All the user facilities relying on binding inference are included into the case tool presented in the next section.

8.3 Integration to CASE tools

These plugins are reusable in any engineering EMF compliant environment and we use them in our proper environment. Figure 22 presents a snapshot of a CASE tool aimed to assist the construction of UML systems with aspectual templates. It was used to experiment the case study presented in the paper using aspectual template libraries such as GOF patterns. This CASE tool includes both a tree-based editor (left side of the figure) and a graphical visualizer of UML models (right side of the figure).

The editor is a specialization of the UML editor provided by Eclipse. Compared to the latter, it adds the capacities to specify new UML aspectual templates, import and bind existing aspectual templates into UML models and process their application. Along these modeling tasks, the aspectual templates and their bindings can be checked at any time using the editor. In addition, the editor includes completion and filtering facilities to avoid or fix errors detected by the set of OCL constraints. These editing facilities rely on the helper plugin. The visualizer provides a graphical view of the model currently edited by the editor or selected in the workspace. This view can help the designer to find interesting aspectual templates before their import or to understand complex aspectual template assemblies and their results.

9 Generalization of the results

In this section, generalization of the results is discussed. We begin by studying the generalization inside UML which is the main topic of the paper. We will see how far aspectual interpretation is applicable to other templates. Then, we further investigate generalization beyond the scope of UML, that is to other forms of model templates.

9.1 Inside UML

All along the paper, we focused on aspectual templates for class models. For this kind of aspectual templates, we formulated a set of constraints that take into account the parameterable elements of class diagrams. While being specific of class models, it can be observed that most of the constraints are related to general dependencies between elements contained in a model such as component/composite dependencies, type dependencies and no-dangling link dependencies. Given that UML provides many other templateable elements than package template and many other parameterables than the ones contained in a class model, two main questions arise. First, how does the notion of aspectual template extend to other templateable elements ? Second, if there are other candidate templateables, how do the presented constraints serve as guidelines to interpret them as aspectual ?

It is worth noting that the current UML specification provides limited information about permitted templates. It only contains examples and explanations for class templates, collaboration templates and package templates based on classes. Other permitted templates are not described in the specification making the task of answering the previous questions not immediate. To help in this study, we proceeded by a systematic introspection of the standard as offered by the UML plugin provided by Eclipse thanks to EMFScript, a reflective scripting language that eases the querying of models and metamodels [41]. This introspection was done by collecting all direct and indirect subclasses of *TemplateableElement* and *ParameterableElement*, then visiting all constituents of templateables to check if they are parameterable ones.

Figure 23 shows the complete sets of parameterable and templateable elements in UML specification. As expected, only a subset of UML elements are parameterable. Concerning templateable elements, they are fewer

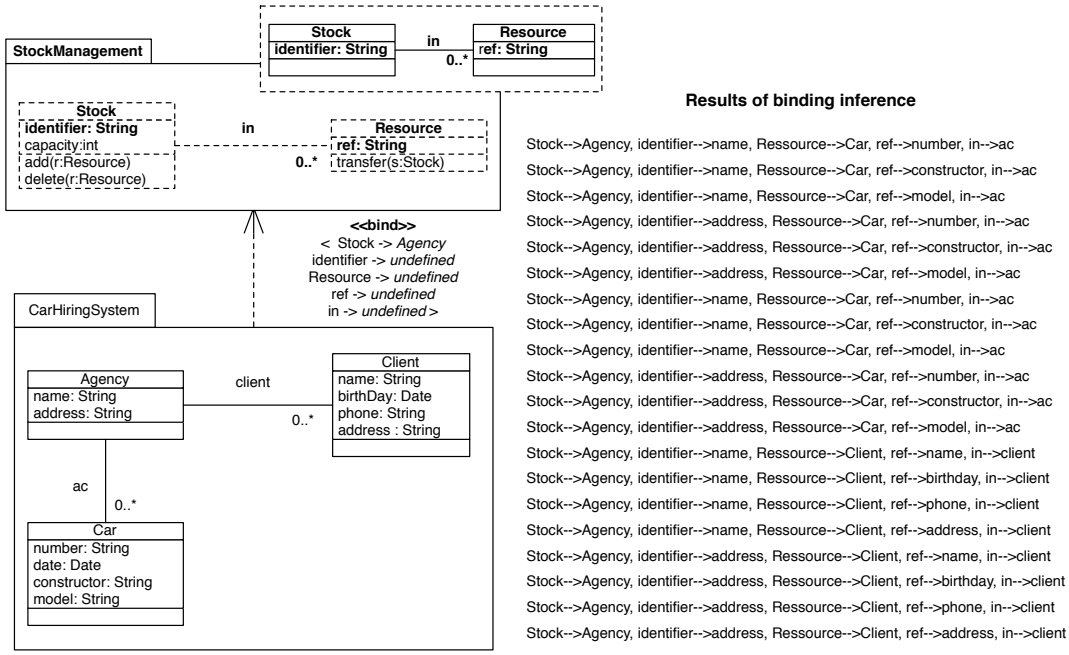


Fig. 21 Binding inference example

than parameterable elements but an important point can be observed comparing them. Although *TemplateableElement* is not a subclass of *ParameterableElement* in the metamodel, all templateable elements are also parameterable elements. Concretely, this means that they can play both roles depending on the modeling situation (see for instance *Class* or *Component* both as template or parameter). In Figure 23, this is shown by including all the templates in the intersection with the set of parameterables. As we can see, there is currently no templateable which is not parameterable.

The identification of parameterable and templateable elements gives a useful knowledge regarding the previous questions. But, for determining if templates can be aspectualized, it is mandatory to know which parameterable elements are involved into which template. For each template, this can be done by determining their parameterable constituents using containment relationships. Table 1 presents the main kinds of model with their root templateable concept and the main corresponding parameterable constituents. From this table we see for instance that a collaboration model (expressed by a collaboration template) can be parameterized by its roles and their types. It is also shown that a use case model can have its constituents (possibly use cases, actors as well as associations between them) as parameters. In addition to the parameterable elements, Table 1 gives the set of elements that are essential for the content of each model kind but are not parameterable in the corresponding template. This allows to observe that some templateables offer limited parameterization capacities. It is the case here for *Interaction*, *Activity* and *StateMachine*.

From the isolation of parameterable elements, it becomes possible to determine if a particular template can be aspectualized. To have this property, a template must have parameters that form a well-structured model. Table 1 shows the result of evaluating this requirement in the "Aspectualizable" column. It appears that only a few templateables have the property. In addition to class model studied in this work, it is also the case for instance, component, use case and deployment models. For each of the corresponding templateables, their set of parameterable constituents is sufficient to form models of their kind. For instance, in a use case template, actors and use cases which are linked by associations are all parameterable elements. Considered together as parameter they form a consistent use case model (even if *Include* and *Extend* relationship are not parameterable). Similarly, components that are parameterable in a component model can be linked via parameterable *Ports* and *Dependency* relationships to form a component parameter model. Similar observations can be made for templates dedicated to instance and deployment models.

Remaining templateables are not aspectualizable because their set of parameterables is not sufficient to form models. This is the case of *Collaboration*, *Interaction*, *Activity* and *StateMachine* templates. For example, in a collaboration template, a role depends on its type and both can be parameters. But connectors between roles which are essential constituents of collaboration are not. This prevents to expose a collaboration model as a parameter and therefore to aspectualize them.

As a conclusion, preceding analysis allows to retain class, instance, use-case, component and deployment mod-

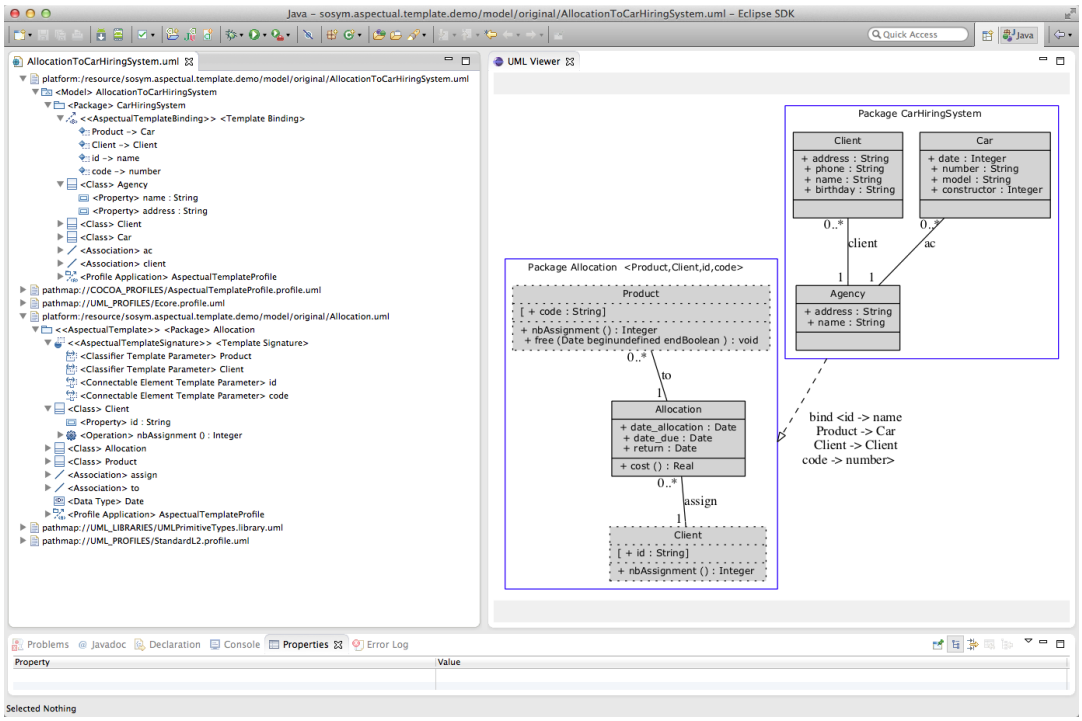


Fig. 22 Using aspectual templates in Eclipse

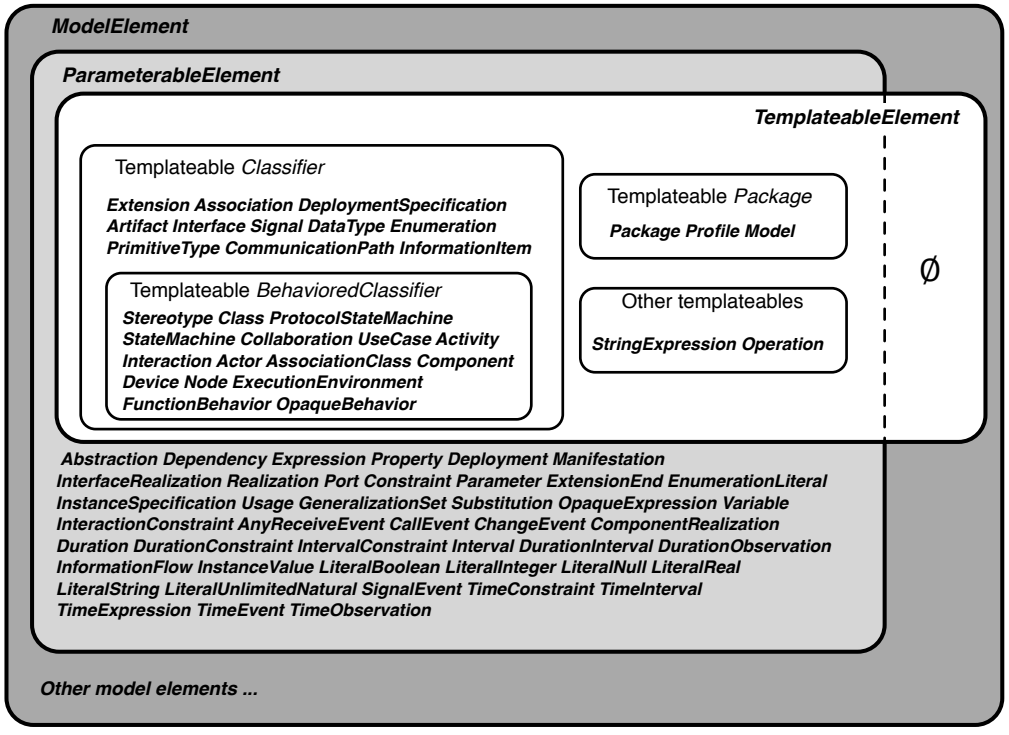


Fig. 23 Parameterable and Templateable elements

Model kind	Templateable root concept	Parameterable constituents	Non-parameterable constituents	Aspectualizable
Class model	Package	Package, Class, Association, Property, Operation, Parameter	Generalization	Y
Instance model	Package	InstanceSpecification, InstanceValue, Type	Slot	Y
Use-case model	Package	UseCase, Actor, Association	Include, Extend	Y
Collaboration model	Collaboration	Property (Role), Type	Connector, CollaborationUse	N
Component model	Package	Component, Port, Property, Dependency, Type	Connector	Y
Interaction model	Interaction	Operation, Port	Lifeline, Message, MessageOccurrenceSpecification, BehaviourExecutionSpecification	N
Activity model	Activity	Operation, Port	ActivityPartition, ActivityEdge, ActivityNode, Action	N
State machine model	StateMachine	Operation, Port	Region, State, Transition	N
Deployment model	Node	Node, Device, Execution Environment, DeploymentSpecification, Artifact, Communication Path, Deployment, Manifestation, Property, Operation		Y

Table 1 Classification of Templateable Models

els for aspectualization. For example, consider aspectualization of a component model in Figure 24. In this example, the aspectual template aims to install a registry of services between two components[8]. The model parameter of this template captures the requirement that these components must be connected through ports having the same interface. Additionally, Figure 24 shows the binding of this template to components which are parts of a management system for hotel rooms. Similar capacities can be obtained for other aspectualizable templates. Concerning deployment models, the work described in [3] presents the specification of reusable deployment patterns as predefined UML templates. It gives some examples of templates parameterized with artifacts which could be easily extended for being aspectual.

For the previous aspectualizable templates, enhancing their aspectual semantics can be achieved by specifying OCL constraints in a way similar to the present work. Similarly to the aspectual template studied here, the constraints for these templates must be defined to handle the specific structure of involved parameters, notably their mutual dependencies which can be related to the ones handled in this paper because they address similar concerns. To give an example, consider the dependency between a port and the owning component when they are parameters in a component model template. It appears that such a dependency is analogous to the one existing between an attribute parameter and its owning class. So constraints defined in the paper are inspiring and may serve as guidelines to define with precision the aspectual interpretation of these templates.

More generally, the fact that parameters across multiple templates have similar dependencies to address brings the factorization of aspectual template constraints into question. At present time, it is not clear whether the definition of general constraints for several aspectual templates is possible. To answer this question, further investigations on the basis of our work are necessary. It seems interesting in particular to study how to exploit the standard class hierarchy of template parameters (*ClassifierTemplateParameter*, *ConnectableTemplateParameter* and *OperationTemplateParameter*) for achieving this generalization.

9.2 Beyond UML

The previous section highlighted that the concepts for supporting UML templates as specified in Figures 2 and 3 appears general enough to be applied to different kinds of model. Indeed, these concepts provide together a complete and generic framework for defining and structuring UML-like template constructs without being necessarily tied to UML concepts. As so, these concepts constitute a "metamodeling pattern" for any template language. Completed with the principles, rules and guidelines presented here for obtaining their aspectual version, this pattern opens the way to the integration of aspectual templates into a wide range of modeling languages. For a particular modeling language, applying the pattern

[8] Such registry of services can be found into component platforms like OSGI in order to facilitate a loose-coupling between components.

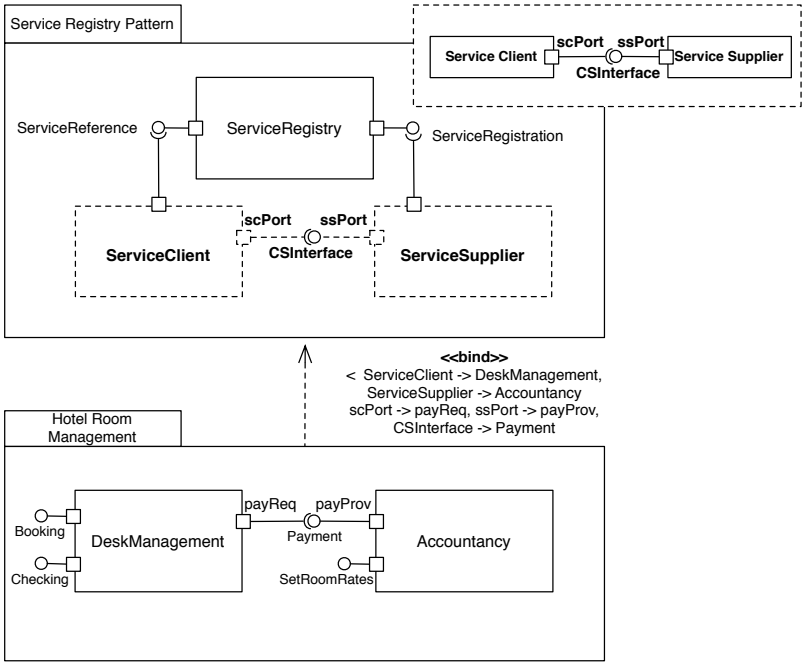


Fig. 24 Aspectual Template of Components

consists in determining the concepts that need to be templateable and their parameterable constituents, then augmenting them with the provided template structure and corresponding derived constraints. Technically, this operation can be done in the metamodel of the language by using inheritance from general template concepts like in UML and [22]. Another way for doing this more modularly is by using aspect-orientation techniques at the metalevel as proposed in [34, 35]. We experimented the application of this pattern to obtain aspectual templates of interaction models similar to simplified sequence diagrams[9]. They were applied in [40] to capture reusable interaction models from the field of convergent telecom applications and construct interaction models of systems in this domain.

Besides its application to specific modeling languages, the notion of aspectual template can also be applied at the metamodeling level. Two works have explored a similar notion of template with the purpose of capturing common patterns or cross-cutting aspects of modeling language definition to design metamodels modularly. [11] presents an aspect-oriented design of a subset of UML by means of package templates. In this work, package templates are similar to Catalysis model frameworks [20] which are parameterized packages. Parameterization is done through string substitution of names. The result of a template instantiation is then merged with elements of a base model according to their names. At the metalevel, such package templates are used by the authors to specify metalevel patterns such as generalization or namespace features and then construct the UML metamodel from the instantiation and merging of these templates.

[18] also applies templates at the metalevel in order to provide an extensible way of defining metamodels in the context of the MetaDepth metamodeling framework. Here, a metamodel template aims to define a generic behavior that can be added to some metamodel to support operations on their models in a particular context like simulation or transformation. The behavior is defined generically by such templates thanks to a "concept" construct which is a separate model expressing both the parameters and a set of structural requirements on these parameters. In some way, this "concept" construct is related to the notion of model as parameter deeply studied in present work as it also aims to provide a higher level of granularity for the template parameters. However, despite this similarity, a "concept" differs fundamentally from model parameter of aspectual templates because it is not part of the template body. More generally, the need for specifying structural requirements of parameterized models is related to model typing. Important works, inspired from the programming world on groups of types [5, 7, 21], were made on the possibility of model substitutability through model (sub-)typing [23, 37]. Though, they focus on the problem of model conformance w.r.t. (sub-)metamodels. Aspectual templates and their binding presented here are mainly concerned with the conformance of candidate models to template parameter models within the same metamodeling space through the question of model inclusion. To investigate this issue, existing works on (meta-)model inclusion and its variants [10, 23, 29] are of interest.

[9] Remind that they are not aspectualizable inside the strict standard.

As far as metalevel is concerned, the present contribution only focused on UML but the classical use of class-based models for metamodel definitions (MOF or EMF) suggests that aspectual templates presented here for UML class diagrams can also be applied at the metalevel. Such an application could enable the construction of metamodels from aspectual templates with the benefits of having a much more powerful and rigorous approach compared to [11,18] mentioned previously. Indeed, they do not provide composition of templates and do not formalize at all the semantic rules for specifying and binding them. Concerning the integration of aspectual templates at the metalevel, that is in the metametamodel, it could be done for instance in a way comparable to the work of [15] which proposes an extension of the Ecore model[10] with UML template constructs in order to get semantic variability in metamodels.

A last area where the results of this work are of interest is Aspect-Oriented Modeling (AOM) which aims to provide model-based aspects typically made of pointcuts and advices. Pointcuts are abstract model elements defining where to affect a base model and the advices are corresponding elements defining how to extend those identified by the pointcuts. In most of the AOM approaches, pointcuts and advices are expressed by two related models with corresponding elements and the model of pointcuts is a subset of the model of advices [28,30,45, 47]. However, the relationship between these two models is most of the time assumed implicitly or loosely defined. This has the consequence that the consistency between the two models is not handled or is only ensured partially. There are several ways to alleviate this problem but the notion of parameterized models like present templates can be a convenient approach in order to get a rigorous definition of the relationship between pointcut and advice models forming an aspect and a way to guaranty their consistency. The idea of relying on parameterized models for a full AOM approach has been explored by the first author in the Smart Adapter approach [31] with the difference that the pointcut part expressed as parameter of the aspect model contains roles which are distinct elements from those of the advice part.

More generally, the goal of moving UML standard towards a complete AOM solution with templates as building blocks is quite interesting but challenging as the standard template formalism has some main limitations: substitution without quantifiers, no wildcards, only addition of elements, no modification, no exclusion. Features of AOM not supported by UML (such as quantification over candidate elements offered by pointcuts or non-monotonic adaptations provided by advices) would imply a deep change in the way standard templates are designed, interpreted and used. The integration of these features raises many questions and calls for a more general study on the capacities of model templates for full AOM.

[10] Note that Ecore has parameterized types but the offered capacities are very close to the generic part of the Java type system.

10 Conclusion

After providing background on the general UML template concept, we concentrated on its specific interpretation for aspectual needs. So called "aspectual templates" and their binding mechanism were defined by enhancing the semantics of standard UML templates in OCL in a fully compatible way: aspectual templates are UML templates. This semantic enhancement constitutes a useful building block for MDE approaches which use UML templates with the requirement that their parameters must form a model, such as pattern, view or aspect oriented ones.

As a major result, due notably to partial binding, the interpretation of the standard allows aspectual template composition in a homogeneous and consistent way: hierarchical construction of richer aspectual templates from the composition of other ones, their capitalization, and finally their usage within modeling assemblies in order to obtain systems.

The paper only retains main ingredients of the general template notion specified in UML, others may be taken into account. We can cite the notion of default value for unbound parameters or the capacity to define a new template by extending an existing one thanks to the concept of *RedefinableTemplateSignature*. How these additional ingredients relate with the aspectual interpretation of templates and may complement this usage is a valuable issue.

For sake of simplicity, this paper only considers basic structural class models. Though, the presented guidelines may extend to other modeling constructs, such as cardinalities of associations, meta-attributes of model elements or inheritance links. This will lead to investigate a richer conformance relationship with specific substitution capabilities in template application. In our previous work on refinement of class models [6], we already studied similar issues and provided solutions specifically for cardinalities and inheritance links. Other works on the merging of class diagrams like [12, 19] would also be a helpful basis to study this issue. More generally, it was shown that works could be made to generalize the results inside and outside UML on the basis of the present work.

Along this paper, we showed how it is possible to build systems as well as "off-the-shelf" rich aspectual templates from the application of multiple ones. This leads to the notion of "model assembly" whose expected properties have been already stated in [33]. Beside aspectual template application, such assemblies may also include other reusing relationships such as standard "merge" and "extends". Beyond model assembling, we saw that

aspectual templates authorize specific model engineering practices such as template extraction, induction, composition and decomposition. All these operators are under study to be integrated in our engineering environment centered on model repository.

Finally, while the precise formalization of aspectual templates in UML using its proper meta-modeling techniques is of value, it was shown that its generalization outside the standard remains a challenge. Much more theoretical investigation is needed, specially on model typing and model inclusion which is a major issue in MDE [23, 29, 37]. As far as model typing is concerned, intuition leads to the idea that the signature of an aspectual template determines a "model type" to which candidate models must conform. We are currently working on this topic by exploiting our previous work on model inclusion and the notion of submodel [10]. All these works should contribute to better theoretical understanding and generalization of aspectual templates for the quest of model reuse.

References

1. MDA. Home Page. http://www.omg.org/mda.
2. W. A. Abed, V. Bonnet, M. Schöttle, O. Alam, and J. Kienzle. Touchram: A multitouch-enabled tool for aspect-oriented software design. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE'12)*, volume LNCS 7745. Springer, 2012.
3. A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel. UML-based Cloud Application Modeling with Libraries, Profiles and Templates. In *Proceedings of CloudMDE Workshop at MODELS'2014*, 2014.
4. J. Bigot and Ch. Pérez. Increasing Reuse in Component Models through Genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, volume 5791 of *LNCS*, pages 21–30. Springer, 2009.
5. Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, March 2003.
6. O. Caron, B. Carré, and L. Debrauwer. Contextualization of OODB Schemas in CROME. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications DEXA 2000*, volume 1873 of *LNCS*, pages 135–149. Springer, 2000.
7. O. Caron, B. Carré, A. Muller, and G.Vanwormhoudt. Programmation par objets structurée en contextes. *Revue L'OBJET, Hermes-Lavoisier*, 13(2-3):11–42, 2007.
8. O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. An OCL Formulation of UML 2 Template Binding. In *Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2004)*, volume 3273 of *LNCS*, pages 27–40. Springer, October 2004.
9. O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. A Coding Framework for Functional Adaptation of Coarse-Grained Components in Extensible EJB Servers. In *47th International Conference Objects, Models, Components, Patterns (Tools'09)*, number 33 in LNBIP, pages 215–230. Springer, 2009.
10. B. Carré, G. Vanwormhoudt, and O. Caron. From subsets of model elements to submodels, a characterization of submodels and their properties. *Software and Systems Modeling*, DOI:10.1007/s10270-013-0340-x, April 2013.
11. T. Clark, A. Evans, and K. Stuart. Aspect-oriented Metamodelling. *The Computer Journal*, 46(5):566–577, 2003.
12. S. Clarke. Extending standard UML with Model Composition Semantics. In *Science of Computer Programming*, volume 44, pages 71–100. Elsevier Science, 2002.
13. S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001,*, pages 5–14. IEEE Computer Society, 2001.
14. S. Clarke and R. J. Walker. Generic Aspect-Oriented Design with Theme/UML. In *Aspect-oriented software development*, pages 425–458. Addison-Wesley Professional, 2004.
15. A. Cuccuru, C. Mraidha, F. Terrier, and G. Sébastien. Templatable Metamodels for Semantic Variation Points. In *Proceedings of Model Driven Architecture- Foundations and Applications*, volume LNCS 4530. Springer Berlin, 2007.
16. A. Cuccuru, A. Radermacher, S. Gérard, and F. Terrier. Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. In *Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*. Springer, 2009.
17. J. de Lara and E. Guerra. Generic Meta-modelling with Concepts, Templates and Mixin Layers. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, MODELS 2010*, volume LNCS 6394. Springer, 2010.
18. J. de Lara and E. Guerra. From types to type requirements: genericity for model-driven engineering. *Software and Systems Modeling*, 12(3):453–474, 2013.
19. J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software and System Modeling*, 7(4):443–467, 2008.
20. D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1999.
21. E. Ernst. Family polymorphism. In *Proceedings of 15th European Conference on Object-Oriented Programming - ECOOP 2001*, volume LNCS 2072. Springer, 2001.
22. I. Garrigos, M. Wimmer, and J-N. Mazon. Weaving aspect-orientation into web modeling languages. In *Current Trends in Web Engineering*, volume 8295 of *LNCS*. Springer, 2013.
23. C. Guy, B. Combemale, S. Derrien, J. Steel, and J.M. Jézéquel. On Model Subtyping. In *Proccedings of 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.
24. J.M. Jézéquel. Model driven design and aspect weaving. *Software and System Modeling*, 7(2):209–218, 2008.
25. S. Kent. Model Driven Engineering. In *Proceedings of the 3rd International Conference on Integrated Formal*

Methods, volume 2335 of *LNCS*, pages 286–298. Springer, May 2002.
26. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241, pages 220–242. Springer, 1997.
27. J. Kienzle, W. Al Abed, F. Fleurey, J-M. Jézéquel, and J. Klein. Aspect-Oriented Design with Reusable Aspect Models. In *Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling*, volume 6210 of *LNCS*, pages 272–320. Springer, 2010.
28. J. Klein, L. Hélouët, and J. M. Jézéquel. Semantic-based Weaving of Scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 27–38. ACM Press New York, NY, USA, 2006.
29. T. Kühne. On model compatibility with referees and contexts. *Software and System Modeling*, 12(3):475–488, 2013.
30. Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, LNCS, pages 498–513. Springer, October 2007.
31. B. Morin, G. Vanwormhoudt, Ph. Lahire, A. Gaignard, O. Barais, and J-M. Jézéquel. Managing variability complexity in aspect-oriented modeling. In *Proceedings of the 11th Internation Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume LNCS 5301, pages 797–812. Springer, 2008.
32. A. Muller. Reusing functional aspects : from composition to parameterization. In *Aspect-Oriented Modeling Workshop - AOM 2004*, Lisbon - Portugal, October 2004.
33. A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. On Some Properties of Parameterized Model Application. In *Proceedings of 1st European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05)*, volume 3748 of *LNCS*, pages 130–144. Springer, November 2005.
34. P. Muller, F. Fleurey, and JM. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proceedings of 8th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2005)*, LNCS 3713, Jamaica, 2005. Springer.
35. G. Perrouin, G. Vanwormhoudt, B. Morin, Ph. Lahire, O. Barais, and JM. Jézéquel. Weaving variability into domain metamodels. *Software and Systems Modeling*, 11(3):361–383, 2012.
36. Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. In *Transaction on Aspect-Oriented Software Development I*, volume 3880, pages 75–105. Springer, 2006.
37. J. Steel and J-M. Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.
38. G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J.M. Bieman. Model composition directives. In *Proceedings of UML 2004 - The Unified Modeling Language. Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 84–97. Springer, 2004.
39. G. Sunyé, A. Le Guennec, and J-M. Jézéquel. Design Patterns Application in UML. In E. Bertino, editor, *Proceedings of 14th European Conference on Object-Oriented Programming (ECOOP'2001)*, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.
40. S. Thiello. Model Templates for Roles Interaction, Master thesis. Technical report, University of Lille, 2010.
41. Ch. Tombelle and G. Vanwormhoudt. Dynamic and Generic Manipulation of Models: From Introspection to Scripting. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *LNCS*, Italy, october 2006.
42. J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
43. UML 2.4.1 Superstructure Specification, 2011. http://www.omg.org/spec/UML/2.4.1/.
44. Auxiliary Constructs Templates, chapter 17. UML 2.4.1 Superstructure Specification, 2011.
45. J. Whittle, K. Praveen, A. Jayaraman, M. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In *Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering*, volume 5560 of *LNCS*, pages 191–237. Springer, 2009.
46. A. Wills. Frameworks and component-based development. In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'96)*, pages 413–430. Springer London, 1997.
47. M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer. A Survey on UML-based Aspect-oriented Design Modeling. In *ACM Computing Surveys*, volume 43, pages 28:1–28:33. ACM, October 2011.