A Demonstration of Compilability for UML Template Instances

José Farinha
ISTAR, ISCTE-IUL, Av. Forças Armadas, Lisbon, Portugal
jose.farinha@iscte.pt

Keywords: UML, Templates, Verification, Compilability, Activities, Software Patterns.

Abstract: Because of the thin set of well-formedness rules associated to Templates in UML, ill-formed elements may result from well-formed bindings to templates. Although such ill-formedness is generally detected by some UML validation rule, the problem is poorly reported if the violated rule does not pertain to the Template construct. Typically, erroneous substitutions of template parameters will be misleadingly reported as compilation problems in the code of operations of the template's instance. This paper demonstrates that a set of well-formedness rules, additional to those of the standard UML, prevents this problem from occurring. Such set of constraints was proposed in a previous paper and named *Functional Conformance* (FC), but a demonstration of its effectiveness was not provided. Such a demonstration is carried out in the current paper adopting UML Activities as the formalism to represent the dynamics of systems and their well-formedness rules as compilability criteria. Carrying out the demonstration revealed further rules than those previously proposed for FC.

1 INTRODUCTION

An UML template is a model element embodying a patterned solution that can be instantiated to solve a recurring problem. A template is instantiated in a model by binding an element of that model to the template, which is done through the *Binding* relationship. In order to have a template instance contextualized to the target model, templates are defined as parameterised elements. A template parameter marks an element participating in the template's specification to tell that it must be substituted by an element of the target model. Only when all of the template's parameters are substituted, it becomes an actual, fully integrated solution in the target model.

In order to ensure that the elements bound to templates are well-formed, UML enforces a set of constraints to parameter substitutions. One such constraint imposes that a substitute element must be of the same kind (Class, Attribute, Operation, etc.) as the parametered element. Another constraint enforces that if a parameter marks a typed element, this element and its substitute have conforming types.

Yet, the set of validations falls short in guaranteeing the well-formedness of template instances. For instance, UML allows an operation *Op1* be substituted by an operation *Op2* whose signature is not compatible with the former's. If *Op1* is substituted by *Op2*, every call to *Op1* in the template's code will be reproduced as a call to *Op2* with an unaligned set of arguments, which is a badly-formed call. Even though the problem was caused by a bad substitution, it will be reported on the operation call, without any back tracking to the source of the problem being recommended by UML (as of version 2.5 (OMG 2015)). There are far more other such scenarios of inadequate substitutions going unnoticed by UML templates and causing incidental errors inside template instances, mainly in the body of the operations. This causes parallax problems in error reporting, a consequence of the scarce set of the validation rules for UML templates.

In (Farinha & Ramos 2015) a set of additional rules was proposed for UML template as a way to overcome the aforementioned problem. Such rules implemented a concept named *Functional Conformance* (*FC*). With such additional rules, improper substitutions of template parameters would be immediately signaled and reported, and error reporting parallax problems removed. Since (Farinha & Ramos 2015) provides only an intuitive perspective on the solution, a formal proof of the effectiveness of it is required.

This paper provides such a proof. One that demonstrates that the original definition of FC (Farinha & Ramos 2015) augmented with two additional constraints ensures the well-formedness of

operations' code resulting from a template binding. In this paper, the well-formedness of operations' code – i.e., methods – is verified assuming that these are represented as UML Activity Diagrams. It is also assumed that methods are purely built with UML Activity's constructs that implement concepts of the traditional Object Oriented Programming, as supported by the most common object-oriented programming languages: Java, C# and C++. More advanced concepts of the UML Activity model – such as Exception Handling, Fork-Join, Signals and Events – are not covered by this paper.

The process of building the proof was useful to uncover the need for two more well-formedness constraints than those suggested by the empirical experimentation that lead to (Farinha & Ramos 2015). This reinforced the importance of developing formal demonstrations. The two additional constraints are related to the preservation of subtyping relationships and of the abstract/non-abstract nature of classifiers when mapping from a template to its instances.

The structure of the paper is as follows: section 2 presents some core concepts of UML templates and introduces the terminology and symbology used in this paper; section 3 briefly presents FC; section 4 includes the demonstration; section 5 presents related work; and section 6 draws some conclusions and foresees further steps towards FC as a sound concept.

2 CONCEPTS, TERMINOLOGY AND SYMBOLOGY

In order to refer to the model fragment that participates in the definition of a template the term "space" is used in this paper. The same term will be used also to refer to the fragment that is covered by an instance of a template. Generally stated, this paper uses "*space* of an element" to refer to the model fragment that is composed of that element and all the elements directly used by it. Hence, the set of model elements composed of a template and all of the elements directly used (referred) by it is called that *template's space*. The term "*template space*" is used for general, non-specific template. Similarly, the term "target space" will be used to denote the model fragment composed of an instance of a template and all the elements used by that instance. In formulas, the template space will be represented by $\mathcal{T}.space$ (a different font is used to differentiate from 'T' representing a type).

Some of the elements in a template's space will be marked as parameters of the template. Others will be used ordinarily by the template, without been specified as parameters.

When binding to the template – i.e., instantiating the template – the elements marked as parameters will be replaced by elements in the target space. This means that the instance of the template – termed the *bound* element – will use those elements of the target space instead of the ones of the template space. The concept in UML representing this replacements in the context of a binding is termed *Substitution*. It is said that an element in the target space *substitutes* an element in the template space.

Being E an element belonging to the space of a template T and B a bind to T, $B.\sigma E$ is the model element in the target space that is the substitute of E in the scope of B. When, in the course of discourse, B is implicit, the substitute of E will simply be referred as σE.

In a binding, the *Projection* of an element E of the template space is the element of the target space that corresponds to E in the context of that binding. I.e., the projection of E is either one of the following:

- σE – i.e. the actual substitute of E – if E is substituted;
- a replica (or reproduction) of E, if E is a member of the template that is not substituted;
- E itself, if E is not substituted nor a template's member (E is simply used by the template; e.g., it is a class used by the template and, therefore, will be used by the bound element as well).

In this paper, the following typing conventions and symbols will be used:

An identifier with a '$^{\mathcal{T}}$', e.g. $E^{\mathcal{T}}$, represents an element in a template space.

An identifier with a '$^{\odot}$', e.g. $E^{\odot}$, represents an element in a target space.

If element E is an element in the template space, then $E^{\odot}$ is the projection of E in the target space.

E –> E' means that element E is substituted by E'. This is the UML notation for substitution.

T' –▷ T means that T' is a subtype of T.

T' =▷ T means that T' is a subtype of T or T itself.

E :▷ T means that E's type is T or a subtype of it.

T ::– f means that T has f as member.

T 👁→ E means that T has visibility on E.

Several UML metamodel's operations are used within formulas. Those that have no parameters are written without parentheses, for clarity reasons.

Finally, there is a term in UML that is worth reminding: *Classifier* is any model element that

represents a classification of instances according to their features. *Classifier* subsumes concepts such as *Class*, *Association*, *Data Type* and *Use Case*.

3 OVERVIEW OF FUNCTIONAL CONFORMANCE FOR UML TEMPLATES

Functional Conformance (FC) is a term that was introduced in (Farinha & Ramos 2015) aiming to denote the equivalence between two model elements, from a third-party, client perspective. It is a directed relationship between two elements e_1 and e_2, herein represented in formulas as 'e_1 **−∗>** e_2', meaning that the first element may be replaced by the second in a model without compromising the consistency of that model. In (Farinha & Ramos 2015) and in the current paper, the concept is applied to the instantiation of UML templates, being proposed as a set of well-formedness constraints that should rule every template parameter substitution. *FC* is defined as a set of criteria, presented in the following subsections.

3.1 Type conformance

Type Conformance (Typ_{Cnf}) states that if an element $e^{\mathcal{T}}$ in the template space has type $T^{\mathcal{T}}$, then the projection of $e^{\mathcal{T}}$ must have the projection of $T^{\mathcal{T}}$ as type.

$$Typ_{Cnf}\ (e^{\mathcal{T}}, e^{\odot}) := \forall\ T^{\mathcal{T}} \in \mathcal{T}.\text{space}, \quad (e^{\mathcal{T}}.\text{type} =\triangleright\ T^{\mathcal{T}}) \Rightarrow (e^{\odot}.\text{type} =\triangleright\ T^{\odot})$$

It should be noted that this criterion should hold for all the types of $e^{\mathcal{T}}$, i.e., for $e^{\mathcal{T}}$'s direct type and for all of its indirect types (ascendants of the direct type). This rule can be announced two-fold, distinguishing when *T* is substituted from when it isn't:

(1) If a type *T* of an element $e^{\mathcal{T}}$ is not substituted, then $e^{\odot}$ must have *T* as type; i.e., $e^{\odot} :\triangleright T$.

(2) If the type *T* of an element e^{T} is substituted, then $e^{\odot}$ must have σT as type; i.e., $e^{\odot} :\triangleright \sigma T$.

3.2 Subtyping conformance

Subtyping Conformance is intended to preserve every *is-a* relationship from the template to the target spaces, in case any classifier substitution occurs on a generalisation hierarchy. The definition is: if $T^{\mathcal{T}}$ is a subtype of $T_{super}{}^{\mathcal{T}}$, then $T^{\odot}$ must be a subtype of $T_{super}{}^{\odot}$ or $T_{super}{}^{\odot}$ itself.

$$Styp_{Cnf}\ (T^{\mathcal{T}}, T^{\odot}) := \forall\ T_{super}{}^{\mathcal{T}} \in \mathcal{T}.\text{space}, \quad (T^{\mathcal{T}} -\triangleright\ T_{super}{}^{\mathcal{T}}) \Rightarrow (T^{\odot} =\triangleright\ T_{super}{}^{\odot})$$

It should be noted that such conformance is required either if both T^{T} and $T_{super}{}^{T}$ are substituted or if only one of them is.

3.3 Multiplicity conformance

Two elements conform regarding multiplicity if they are both single-valued (multiplicity's upper bound = 1) or both multivalued (multiplicity's upper bound > 1) and, in the latter case, if they are both ordered or both not-ordered:

$$Mlt_{Cnf}\ (e^{\mathcal{T}}, e^{\odot}) := e^{\mathcal{T}}.\text{isMultivalued} = e^{\odot}.\text{isMultivalued} \wedge e^{\mathcal{T}}.\text{isOrdered} = e^{\odot}.\text{isOrdered}$$

3.4 Contents conformance

Contents Conformance (Cts_{Cnf}) applies only to model elements that are namespaces. In the context of a certain bind, the namespace $ns^{\odot}$ conforms in contents with $ns^{\mathcal{T}}$ if every member of the $ns^{\mathcal{T}}$ being used by the template is substituted by a member of $ns^{\odot}$. If the namespace is a type, its members are properties, operations, or inner types. If it is package, members are packages or classifiers. Figure 1 shows an example with classifiers where Cts_{Cnf} doesn't hold: class *Cp* may not be substituted by *Cs1*, because *Cs1* doesn't provide a substitute for attribute *y*. On the contrary, in Figure 2 *Cs1* conforms in contents with *Cp*.

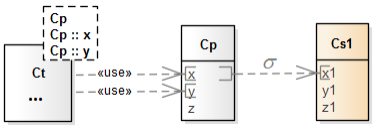

Figure 1: Example of lack of contents conformance.

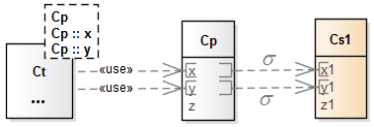

Figure 2: Example of contents conformance.

Cts_{Cnf} is formulated for the namespaces $ns^{\mathcal{T}}$ and $ns^{\odot}$, for a binding to a template *T*, as:

$$Cts_{Cnf}\ (ns^{\mathcal{T}}, ns^{\odot}) := \forall e^{\mathcal{T}} \in ns^{\mathcal{T}}.\text{members} \cap \mathcal{T}.\text{space}, \quad \exists\ e^{\odot} \in ns^{\odot}.\text{elements}: e^{\mathcal{T}} -> e^{\odot}$$

Cts_{Cnf} has a corollary and a specialisation, which are presented in the following subsections.

3.4.1 Membership conformance

An element $e^{\odot}$ conforms in membership to an element $e^{\mathcal{T}}$ if at least one of its namespaces substitutes one of $e^{\mathcal{T}}$'s namespaces. *Membership Conformance* (Msh_{Cnf}) enforces that, if A is substituted by B, members of A must be substituted by members of B.

$$Msh_{Cnf}(e^{\mathcal{T}}, e^{\odot}) := (e^{\mathcal{T}}.namespaces)^{\odot} \cap e^{\odot}.namespaces \neq \emptyset$$

It is possible to demonstrate that Msh_{Cnf} may also be formulated as:

$$\forall NS^{\mathcal{T}}, (NS^{\mathcal{T}} ::- e^{\mathcal{T}}) \Rightarrow (NS^{\odot} ::- e^{\odot})$$

3.4.2 Signature conformance

Signature Conformance (Sig_{Cnf}) is a specialisation of Cts_{Cnf}, as applied to operations. It is the criteria that ensures that a substituting operation has a set of parameters compatible with that of the substituted.

Assuming that the input (*in* and *inout*) and output (*out* and *inout*) parameters of an operation are given by *inParams()* and *outParams()*, respectively, Sig_{Cnf} is formulated for the operations $op^{\mathcal{T}}$ and $op^{\odot}$ as:

$$\begin{aligned} Sig_{Cnf}(op^{\mathcal{T}}, op^{\odot}) := \\ & op^{\mathcal{T}}.parameters.size = op^{\odot}.parameters.size \wedge \\ & (\forall i \in [1, op^{\mathcal{T}}.inParams.size], \\ & \quad op^{\mathcal{T}}.inParams_i \; -\!\!*\!\!> \; op^{\odot}.inParams_i) \wedge \\ & (\forall i \in [1, op^{\mathcal{T}}.outParams.size], \\ & \quad op^{\mathcal{T}}.outParams_i \; <\!\!*\!\!- \; op^{\odot}.outParams_i) \end{aligned}$$

Notice that conformance among input parameters is enforced from template to target while among output and return parameters is the opposite.

The verification of FC between operation parameters is done through:

$$\begin{aligned} opPrm^{\mathcal{T}} \; -\!\!*\!\!> \; opPrm^{\odot} := \; & Typ_{Cnf}(opPrm^{\mathcal{T}}, opPrm^{\odot}) \wedge \\ & Mlt_{Cnf}(opPrm^{\mathcal{T}}, opPrm^{\odot}) \end{aligned}$$

3.5 Staticity conformance

This criterion establishes that a static feature may only be substituted by another that is also static, and a non-static by a non-static:

$$Stc_{Cnf}(f^{\mathcal{T}}, f^{\odot}) := f^{\mathcal{T}}.isStatic = f^{\odot}.isStatic$$

3.6 Abstraction conformance

This criterion applies only to parametered elements that are classifiers and is already supported by UML 2.5. It states that a classifier that is not abstract may only be substituted by another classifier that is not abstract as well:

$$\begin{aligned} Abst_{Cnf}(C^{\mathcal{T}}, C^{\odot}) := \\ & \neg C^{\mathcal{T}}.isAbstract \Rightarrow \neg C^{\odot}.isAbstract \end{aligned}$$

3.7 Visibility requirement

This is a requirement that enforces that an element may substitute a template parameter only if that element is visible from the bound element. It may be formulated for a binding *B* and element *e* as:

$$B.boundElement \; \multimap \; e^{\odot}$$

3.8 Functional conformance enforcement

If FC is enforced as a validation rule of the UML concept of *template substitution*, FC holds between any element *E* in the template space and its projection, since for every form of $E^{\odot}$ one of the following will hold:

$E \; -\!\!*\!\!> \; \sigma E$, by well-formedness constraint;

$E \; -\!\!*\!\!> \; \rho E$, by replication; ρE is a reproduction of *E*, according to the semantics of the Binding relationship.

$E \; -\!\!*\!\!> \; E$, trivially.

It shall be proved that $E \; -\!\!*\!\!> \; E^{\odot}$ is enough to ensure the well-formedness of the bound element's operations if those operations are operations are originally well-formed in the template.

4 DEMONSTRATION

4.1 Strategy

4.1.1 Representing Code by UML Activities

The goal of this demonstration is to show that the code of operations in a class template remains compilable once reproduced in instances of that template. For instance, a class template for keeping a list of items ordered by name would have an operation *insert (Item)* with the following Java definition:

```
AlphabeticList::insert (Item itm) {
   int k = 1;
   for (itm.name < self.items[k].name)
         k++;
   self.items.insertAt (itm, k);
}
```

If a class *CustomerList* is bound to the template *AlphabeticList*, substituting class *Item* by *Customer* and *itm* by *cust*, the following method would be generated:

```
CustomerList::insert (Customer cust) {
   int k = 1;
   while (cust.name < self.items[k].name)
         k++;
   self.items.insertAt (cust, k);
}
```

It should be proved that if method *AlphabeticList::insert (Item)* compiles successfully and FC is enforced on substitutions, then *CustomerList::insert (Customer)* compiles as well. However, instead of using the syntax rules of a programing language to check compilability, it will be assumed that all code is represented by UML Activity Diagrams and compilability will be checked using the well-formedness rules for Activities. E.g., the previous method is considered equivalent to the activity in Figure 3.

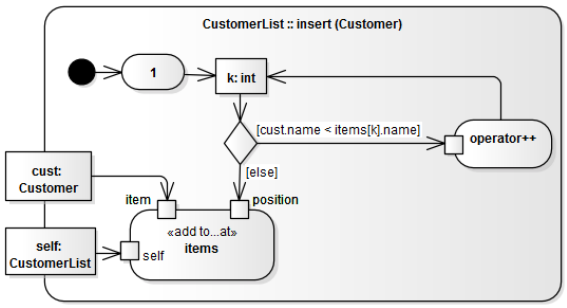

Figure 3: A method represented as an activity with an expression.

It could be noted however that the compilability of the method in Figure 3 may not yet be fully assessed by UML Activity well-formedness rules. That's because one of the guards of the decision node is expressed as an expression. The assessment of that expression would require UML's well-formedness rules for expressions as robust as those of programing languages, which is not the case: the UML metamodel stores expressions as simple tree structures, without establishing validation rules for the compatibility between those trees' nodes. E.g., UML considers *3 * "potato"* a valid expression. Hence, to achieve our goals, expressions must be represented as activities. E.g., the guard expression in Figure 3 must be replaced by the composite activity *cust.name < items[k].name* shown in Figure 4, which internally should be as in Figure 5. Since this expression-activity feeds the «decisionInputFlow» of the decision node, its result will steer execution as desired. Once every expression is formally represented by an activity, the compilability of a method may be fully verified through UML Activity well-formedness rules.

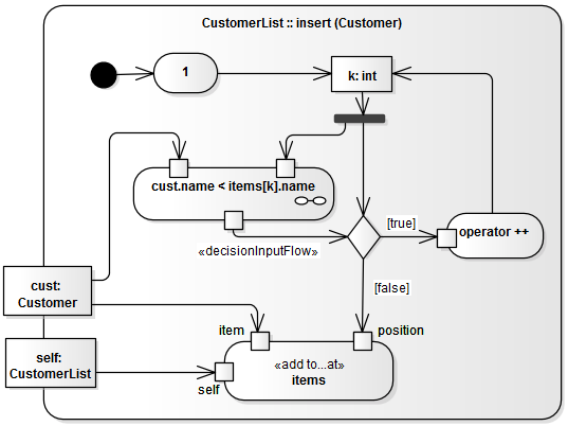

Figure 4: A method fully represented as an activity.

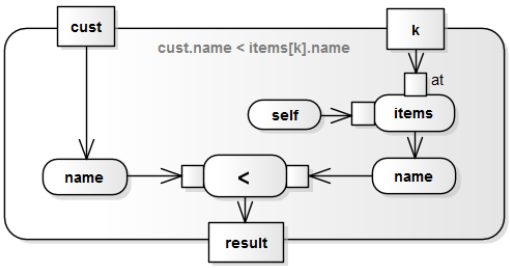

Figure 5: The internals of an expression-activity.

For the sake of a clear scope definition, the following concepts of the UML Activity formalism and their specializations are considered sufficient to represented structured programing code with object orientation, as support by Java, C# and C++: *Object Action*, *Structural Feature Action*, *Call Action*, *Object Node*, *Control Flow*, *Object Flow*, and *Decision Node*. Every construct of such languages that is not are subsumed by those concepts is, therefore, out of the scope of this paper. A proper paper would be required to demonstrate the compilability of programs that use such constructs.

The problem of compilability assessment will be further reduced to the assessment of the well-formedness of a general action, as will be shown in section 4.1.3.

4.1.2 UML Activities

Paraphrasing (OMG 2015, sec.15.1): "An Activity is a kind of behaviour that is specified as a graph of nodes interconnected by flows. A subset of the nodes are executable nodes that embody lower-level steps in the overall activity." Such executable nodes are called *Actions* and correspond to statements in programing languages. "*Object Nodes* hold data that is input to and output from executable nodes", and may represent variables, operation parameters or their arguments. The data in object nodes moves across *Object Flow*s. The sequencing of actions is specified through *Control Flows*, and these may be controlled by *if-then-else*, *switch*, *fork-join* or *loop* structures, globally designated *Control Nodes*. In this paper, it is considered that the operations' code under consideration is purely represented using the UML concepts described below.

Object Actions operate on objects as a whole, representing statements that create objects (`new MyClass`), destroy them (`delete myObj`), check their classification (`myObj instanceof MyClass`) or their identities (`obj1 == obj2`). Examples in Figure 6.

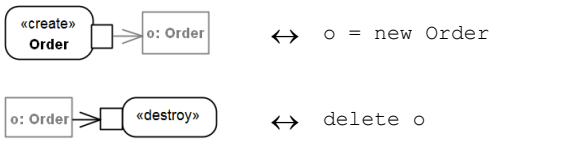

Figure 6: Object actions.

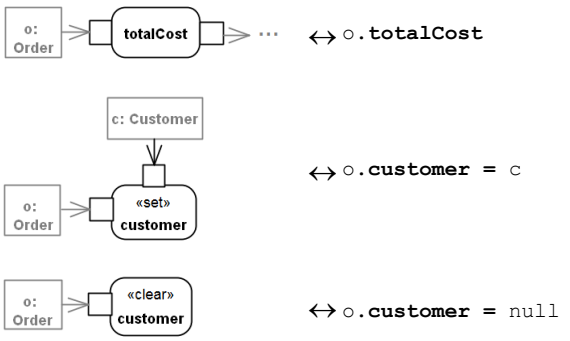

Figure 7: Structural Feature actions.

Object actions also include *Value Specification Actions*. These are actions that yield a value after evaluating a textual expression, including those with a single literal. In this paper, only actions evaluating literals are considered (such as the '1' action in Figure 4). Other *value specification actions* are represented by composite activities, as mentioned in section 4.1.1.

Structural Feature Actions read or write on properties of objects (Figure 7).

A *Call Action* invokes a behaviour (Figure 8) or an operation on an object (Figure 9).

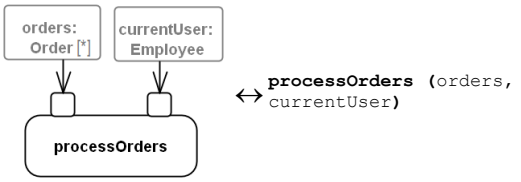

Figure 8: 'Call Behaviour' action.

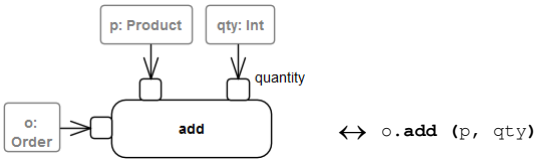

Figure 9: 'Call Operation' action.

Object Nodes are used to store data that is used and/or produced by actions. Those may represent variables (e.g., '*o: Order*' in the examples above) or, through the concept of *Pin* – a specialization of *Object Node* – may represent behaviour's or operation's parameters (e.g., *quantity*, in Figure 9).

A *Decision Node* chooses one between multiple outgoing flows: the first one whose guard is true. Figure 10 shows two possible configurations for a decision node. Decision nodes may also be used to implement loops, as shown in Figure 11. Even though UML provides a construct specific for looping (*LoopNode*), it is enough to consider decision nodes for compilability assessment.

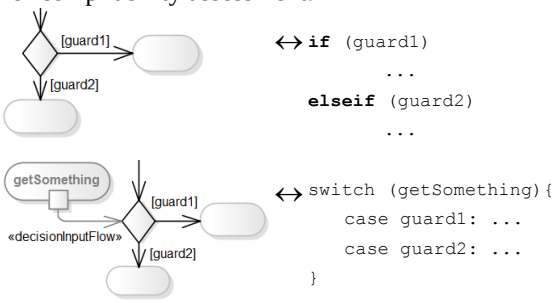

Figure 10: Decision nodes.

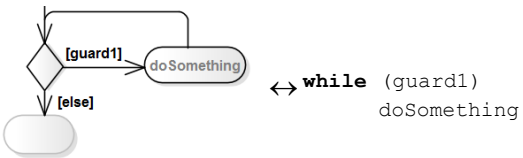

Figure 11: A loop in an activity.

4.1.3 Demonstrating the Well-formedness of Activities through an Archetypal Action

To keep the demonstration simple, it is adequate to narrow down the set of elements whose compilability must proved. To achieve this, first we filter out those elements whose compilability is not affected by template binding. Secondly, the remaining elements are reduced to a simpler, common representation.

Taking into account the semantics of UML template binding – recalling: the bound element is a replica of the template with superimposed substitutions – and that the compilability of the template is a premise, it may be deduced that only those elements being impacted by substitutions may spoil the compilability of the bound element. This narrows down the set of elements to consider those whose validation rules reference parameterable (therefore, substitutable) elements. Since neither the source nor the target of activity flows are parameterable, flows' connecting points are never changed by template substitutions. This means that the topology of an activity is preserved from the template to the bound element. Consequently, it is possible to consider individually each element kind presented in the previous section.

Control Flow metadata and constraints exclusively deal with topology, except in two aspects: the flow's weight and guard. Since the concept of *weight* doesn't exist in programing languages, control flows may only jeopardize compilability because of their guards. As seen in the previous sections, all expressions that are not single literals are represented as composite activities. This means that the compilability of a guard may be ultimately determined by the joint compilability of an activity without guards and a fragment such as the one in Figure 12. The same is also valid for Decision Nodes, if we substitute in Figure 12 "resultFromExpression" by "resultFromDecisionInputFlow", and "true: Boolean" by "aLiteral: T2" or "aVariable: T2". This filters out guards, control flows and decision nodes from consideration.

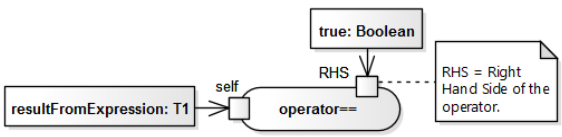

Figure 12: Fragment of the semantics of a guard.

Hence, the assessment of the compilability of a bound activity becomes reduced to the verification of the action kinds shown in the previous section and of that in Figure 12, in both cases taking into account the objects and object flows that connect to those actions. This allows further simplifying our demonstration by subsuming all those actions to a common representation: the generic, archetypal action in Figure 13. Compilability will then be verified by formulating UML well-formedness rules exclusively in terms of that action. The archetypal action shown in Figure 13 aims at representing a feature call in a broad sense: a call to a feature of an object, of a collection of objects (e.g., a call to *size()*), of a class (a call to a static feature), or of the run-time system (e.g., a call to the *new* operator). The demonstration strategy from this point on is somewhat straightforward: assuming that well-formedness rules hold for the archetypal action in a template, it must be shown that they hold as well for the corresponding bound action if FC is enforced in the binding. I.e., representing the archetypal action by *a*, it should be shown that:

$$\text{WellFormed}\,(a^{\mathcal{T}}) \wedge (a^{\mathcal{T}} -\!\!*\!\!> a^{\odot}) \Rightarrow \text{WellFormed}\,(a^{\odot})$$

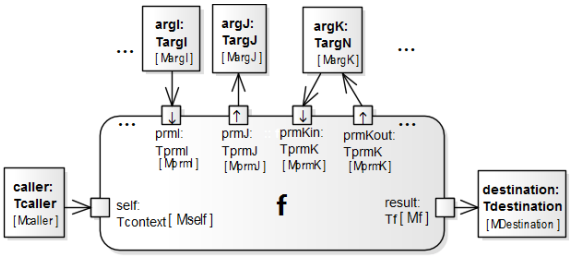

Figure 13: An action in the template.

It is considered that the archetypal action is defined within a template and, therefore, is reproduced in every element bound to that template. The archetypal action within the template will be referred interchangeably as *templated action* and represented as in Figure 13 (without '$^{\mathcal{T}}$'). Its reproduction in a bound element will be referred as *bound action* and represented as in Figure 16.

4.1.3.1 The Templated Action

The feature being called by the templated action in Figure 13 is represented by the meta-variable ***f***. For *Create Object* actions *f* is a class, not a feature. For *Value Specification* actions *f* is an expression; specifically to this paper, it is a literal expression. For *Destroy Object* actions *f* doesn't exist.

Self is a pin that represents the usual variable *self*/*this*: a reference to the object that executes the feature, from the perspective of the code of that feature. *Self* doesn't exist in *Create Object*, *Value Specification*, and *Call Behaviour* actions.

As imposed by UML's constraints (OMG 2015, sec.16.14.54.6 and 16.14.10.6), *self*'s type is the type that owns – i.e., declares and provides context to – the feature being called. The type of *self* is identified by the meta-variable ***Tcontext***.

The multiplicity of the *self* pin is represented by ***Mself***. *Mself*'s upper bound may be either 1 or greater than 1. It must be 1 if the feature being called (*f*) is structural (*f* is attribute or association end). If *f* is an operation, *self* may be multivalued (multiplicity's upper bound > 1). This is required to support calls to collection operations (*size()*, *includes(…)*, etc.).

The ***caller*** object node represents the instance that embodies *self* in an execution of the action. In a statement 'anObject.feature', *anObject* is represented in Figure 13 by *caller*. Depending on the topology of the activity containing the action, *caller* may represent a variable, a parameter of the activity that contains the action (Figure 14) – including that activity's *self* – or the *result* pin of a preceding action (Figure 15). *Caller*'s type and multiplicity are represented by the metavariables ***Tcaller*** and ***Mcaller***, respectively.

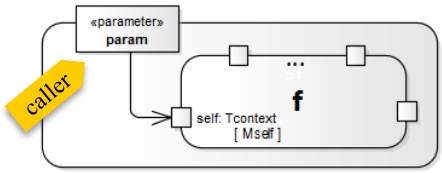

Figure 14: *Caller* is a parameter of the owning activity.

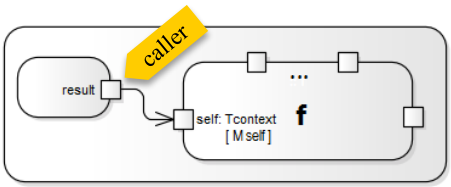

Figure 15: *Caller* is a previous *result*.

***Prm**i*, for *i* from 1 to *N*, is a parameter of *f* with direction other than 'return'. *Prm*$_i$'s type and multiplicity are represented by the meta-variables ***Tprm**i* and ***Mprm**i*, respectively. *Prm*$_i$ also represents the pin that passes values to or from the *prm*$_i$ operation parameter, depending on that parameter's direction being *in* or *out*, respectively. If *prm*$_i$ is a bidirectional parameter (*inout*), values may be passed to and from it. Since UML pins may not be bidirectional, two pins are required to every *inout* parameter: these will be called *prm*$_{i\_in}$ and *prm*$_{i\_out}$. That's the case of *prm*$_K$ in Figure 13. According to UML ((OMG 2015), pp. 493-4), in feature call actions, pins may have characteristics different from those of the corresponding feature parameters, as long as they are consistent. However, since structured programming languages don't have a concept such as *Pin*, this document assumes that every *prm*$_i$ pin is a pure surrogate of the corresponding *prm*$_i$ parameter, being the pin's characteristics (type, multiplicity, etc.) derived from those of the parameter.

***Arg**i*, for *i* from 1 to *N*, is the argument passed to *prm*$_i$. *Arg*$_i$'s type and multiplicity are represented by ***Targ**i* and ***Marg**i*, respectively. Similarly to *caller*, *arg*$_i$ may represent a variable, a parameter of the activity, or the *result* pin of an upstream action.

Result is the pin that yields the value returned by the action. If *f* is a property, *result* yields the value of that property in the instance provided by *caller*. If *f* is an operation, *result* represents that operation's parameter whose direction is 'return', i.e., it yields the value returned by the operation. *Result´s* type and multiplicity are represented by ***Tf*** and ***Mf***, respectively. As above, although UML allows a pin and the corresponding parameter be different, it will be assumed that the *result* pin's type and multiplicity are always in sync with those of *f* – suggestively, that's why those are named "*Tf*" and "*Mf*".

Destination represents the element that receives the result of the action. Also depending on the topology of the activity containing the action, it might be a variable, an output (*out*, *inout* or *return*) parameter of the activity, or a pin of a downstream action (a subsequent *self* or *prm*$_i$).

4.1.3.2 The Bound Action

The reproduction of the archetypal action within the bound element will be termed *bound action* and it will be as in Figure 16. In that figure, the elements that may differ from their original counterparts are marked with '⊙' (in some cases reduced to a '○', due to typewriting constraints).

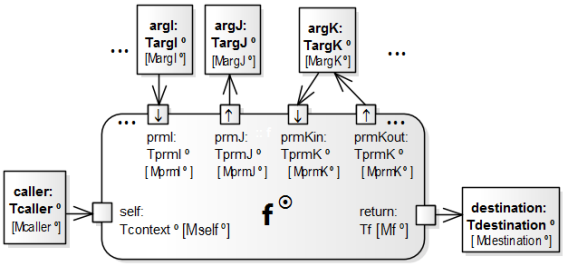

Figure 16: The bound action.

4.2 Compilability criteria and its demonstration for the bound action

This section defines how compilability is assessed in the demonstration. This will be done using UML Activity well-formedness constraints: if these hold for an activity, that activity is compilable. Only the following constraints are relevant:

- Those whose formulation includes elements that UML defines as parameterable in a template (therefore, substitutable in a binding).
- Those verifying the kinds of elements existing in the action archetype, which are: *Object Node* and *Pin* (Figure 17); Object Flow (Figure 18); *Structural Feature Action*, *Call Action* and *Object Action* (Figure 19);

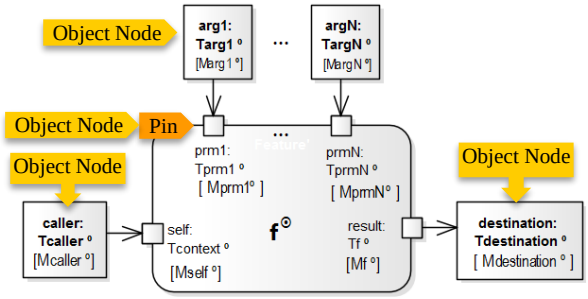

Figure 17: Object Nodes in the action archetype.

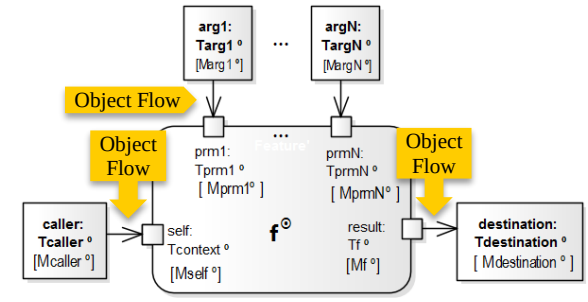

Figure 18: Object Flows in the action archetype.

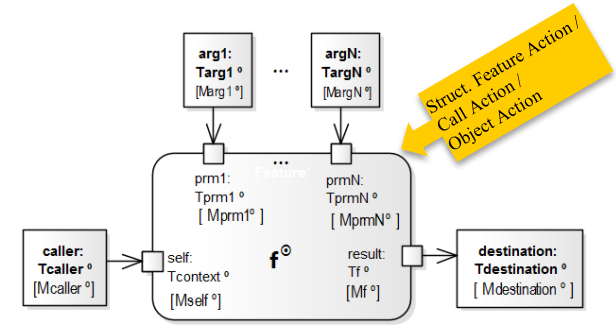

Figure 19: Kinds allowed for the action archetype.

The following subsections summarise the constraints in UML 2.5 (OMG 2015) that are relevant to this demonstration and formulate them in terms of the archetypal action. Each constraint is marked with the bullet '🔦'. The holding of those constraints in the templated action will be premises. The holding in the bound action are the hypothesis that must be proved on the basis of those premises and that FC also holds between every element and its projection in the target space. Premises are identified by numbers prefixed with a 'P'.

4.2.1 On Object Flows

🔦 *Compatible_types*: "(…) the downstream object node type must be the same or a supertype of the upstream object node type" (OMG 2015, p.427). I.e., designating the upstream and downstream object nodes' types as T_{from} and T_{to}, respectively: $T_{from} =\triangleright T_{to}$. This provides the premise: $\forall(T_{from}, T_{to}) \in \{(Tcaller, TContext), (Targ_{i\_in}, Tprm_{i\_in}), (Tprm_{j\_out}, Targ_{j\_out}), (Tf, Tdestination)\}$, $T_{from} =\triangleright T_{to}$ (P1).

It must be proved that, for the same pairs (T_{from}, T_{to}): $T_{from}^{\odot} =\triangleright T_{to}^{\odot}$. Which is ensured by *Subtyping Conformance*: (P1), $Styp_{Cnf}(T_{from}, T_{from}^{\odot}) \vdash T_{from}^{\odot} =\triangleright T_{to}^{\odot}$. QED.

🔦 *Same_upper_bounds*: "Object nodes connected by an object flow (…) must have the same upper bounds" (OMG 2015, p.427). This is a rule that must be redefined, because it originally aims at ensuring semantic compatibility, not compilability. To ensure compilability, it is enough that object nodes connected by a flow are both single-valued or both multivalued. Hence, designating the object nodes as *from* and *to*: from.isMultivalued = to.isMultiValued. This leads to the premise: $\forall$(from, to) $\in$ {(caller, self), (arg_i, prm_i), (result, destination)}, from.isMultivalued = to.isMultivalued (P2).

It must be proved that: from$^{\odot}$.isMultivalued = to$^{\odot}$.isMultivalued (2). Considering Mlt_{Cnf} : (P2), ($\forall$from $\in$ {caller, arg_i, result}, Mlt_{cnf} (from, from$^{\odot}$)), ($\forall$to $\in$ {self, prm_i, destination}, Mlt_{cnf} (to, to$^{\odot}$)) $\vdash$ (2). QED.

4.2.2 On Structural Feature Actions

Such actions access a property in order to read, write or clear it (set it to null).

🔦 *Multiplicity*: "The multiplicity of the *self* input pin must be 1..1" (OMG 2015, sec.16.14.54). The equivalent of this constraint in a programming

language would be an enforcement that any expression 'obj.feature' should be preceded by the assertion 'obj != null'. This is clearly a rule for semantic equivalence, not for compilability. Therefore, this constraint must be redefined: *self* must be 0..1. In terms of the templated action: ¬ self.isMultivalued (P3).

It must be proved that: ¬ self$^{⊙}$.isMultivalued (3).

Considering *Multiplicity Conformance*: (P3), Mlt_{Cnf} (self, self$^{⊙}$) ⊢ (3). QED.

Not_static: "The structural feature must not be static" (OMG 2015, sec.16.14.54). In terms of the templated action: ¬ f.isStatic (P4).

Staticity Conformance ensures that the constraint also holds in the target space: (P4), Stc_{Cnf} (f, f$^{⊙}$) ⊢ ¬ f$^{⊙}$.isStatic. QED.

Object_type: "The structural feature must either be an owned or inherited feature of the type of the *object* input pin, (…)". I.e.: Tcontext ::– f (P5).

Msh_{Cnf} (page 4) ensures that the constraint holds in the target space: (P5), (1) ⊢ Tcontext$^{⊙}$ ::– f$^{⊙}$. QED.

Visibility: "The visibility of the structural feature must allow access from the object performing the action". It should be noted that the object in this citation is the one executing the whole activity that contains the action. Its type may be get from an action by means of '.*containingActivity.context*'. Thus, in terms of the templated action *a*, this constraint is formulated as: a.containingActivity.context 👁→ f.

It must be proved: a$^{⊙}$.containingActivity.context 👁→ f$^{⊙}$ (4). It worth reminding that the templated action is part of an activity belonging to the template under consideration. From the semantics of Binding, *a*$^{⊙}$ belongs to an activity belonging to the bound element. I.e., representing the bound element as *BE*: a$^{⊙}$.containingActivity.context = BE (5). Hence: (4), (5) ⊢ (BE 👁→ f). Since this is the formulation of the *Visibility Requirement*, (4) holds. QED.

The remaining well-formedness constraint, *one_featuring_classifier*, is ensured by that fact that the template action is well-formed (a premise).

4.2.3 On 'Read Structural Feature' Actions

Multiplicity: "The multiplicity of the structural feature must be compatible with the multiplicity of the *result* output pin" (OMG 2015, sec.16.14.42.5). I.e.: f.compatibleWith (result). According to (OMG 2015, sec.7.8.8.7), 'compatibleWith ()' means that the multiplicity of *f* must be comprehended by that of the *result* pin. However, due to the goal and scope of this paper, it will be considered that to ensure compilability it is enough that *f* and *result* are both single-valued (multiplicity = 1) or both multivalued (multiplicity > 1). Therefore, this constraint will be redefined as: f.isMultivalued = result.isMultivalued (P6).

It must be proved that: f$^{⊙}$.isMultivalued = result$^{⊙}$.isMultivalued (6).

Considering *Multiplicity Conformance*: (P6), Mlt_{Cnf} (f, f$^{⊙}$), Mlt_{Cnf} (result, result$^{⊙}$) ⊢ (6). QED.

Type_and_ordering: "The type and ordering of the *result* output pin are the same as the type and ordering of the structural feature" (OMG 2015, sec.16.14.42.5). It is likely there is a lapse in this definition, because there is no reason for requiring that *f* and *result* have exactly the same type, instead of allowing *f*'s return type be a subtype of *result*'s type. Since it provides greater flexibility and doesn't compromise compilability, we will assume that the intended formulation is: f.type =▷ result.type ∧ f.isOrdered = result.isOrdered (P7).

Two hypothesis must be proved: f$^{⊙}$.type =▷ Mf$^{⊙}$ (7); f$^{⊙}$.isOrdered = result$^{⊙}$.isOrdered (8).

The first one tells that if *f* is substituted, σ*f*'s type is the projection of *Mf* or a subtype of it. Since f.type = Mf, (7) may be rewritten as: f$^{⊙}$.type =▷ (f.type)$^{⊙}$. Since this is the formulation of Typ_{Cnf} for the projection of *f*, if Typ_{Cnf} holds, (7) holds as well.

Regarding the second hypothesis: since the ordering of a model element is not parameterable by itself in a template, *result*$^{⊙}$ is as ordered as *result*. Therefore, (8) may be written as: f$^{⊙}$.isOrdered = result.isOrdered (9).

(P7), (9) ⊢ f$^{⊙}$.isOrdered = f.isOrdered. Since this expression is included in the formulation of Mlt_{Cnf} (f, f$^{⊙}$), if Mlt_{Cnf} holds, (8) holds as well. QED.

4.2.4 On 'Write Structural Feature' Actions

Such actions set the value of an object property (second example in Figure 7). It corresponds to an archetypal action with a single *prm* pin: prm_1.

Type_of_value: "The type of the *value* input pin must conform to the type of the structural feature" (OMG 2015, sec.16.14.62.6). UML defines conformance of a classifier C1 to classifier C2 as: C1 =▷ C2 (OMG 2015, sec.9.9.4.7). The *value* pin referred in the citation corresponds to prm_1 in the archetypal action. Therefore, this constraint's formulation is: prm_1.type =▷ f.type. Which may be written as: prm_1.type =▷ Tf (P8).

It must be proved that: (prm_1.type)$^{⊙}$ =▷ (f$^{⊙}$.type)$^{⊙}$. Which may be written as: T$prm_1^{⊙}$ =▷ Tf$^{⊙}$.

Considering *Subtyping Conformance*: (P8), $STyp_{Cnf}$ ($Tprm_1$, $Tprm_1^{\odot}$) ⊢ ($Tprm_1^{\odot}$ =▷ $Tf^{\odot}$). QED.

Multiplicity_of_value: "The multiplicity of the *value* input pin is 1..1" (OMG 2015, sec.16.14.62.6). The formulation and demonstration of this constraint are the same as those of the *Multiplicity* constraint in section 4.2.2, replacing the *self* pin by prm_1 pin.

4.2.5 On Call Actions

Call Actions encompass those that invoke a behaviour or an operation.

Argument_pins: "The number of argument input pins must be the same as the number of input (in and inout) parameters of the called behaviour or operation. The type, ordering and multiplicity of each argument Input pin must be consistent with the corresponding input parameter" (OMG 2015, sec.16.14.8.7). I.e., denoting the list of an action's input pins by *inPins*:

inPins.size = f.inputParameters.size ∧
(∀i ∈ [1, inPins.size]:
 $inPin_i$.type =▷ f.$inputParameter_i$.type ∧
 $inPin_i$.isOrdered = f.$inputParameter_i$.isOrdered ∧
 $inPin_i$.isMultivalued =f.$inputParameter_i$.isMultivalued) (P9)

The verification of this constraint on the bound action checks that, in case *f* is substituted, σ*f* has a set of parameters that remains aligned with the sequence of input pins. The demonstration of this constraint (and the next) consists in showing that the sequence of input (output) parameters of σ*f* (or $f^{\odot}$) is aligned with the sequence of input pins of the templated action, which happens to be the same as the bound action's, since binding to a template doesn't change the topology of an activity diagram (see section 4.1.3). Therefore, it must be proved that:

inPins.size = $f^{\odot}$.inputParameters.size ∧
(∀i ∈ [1, inPins.size]:
 $inPin_i$.type =▷ $f^{\odot}$.$inputParameter_i$.type ∧ (10)
 $inPin_i$.isOrdered = $f^{\odot}$.$inputParam_i$.isOrdered ∧
 $inPin_i$.isMultivalued = $f^{\odot}$.$inputParam_i$.isMultivalued)

(P9) ∧ (10) ⇔
f.inputParameters.size = $f^{\odot}$.inputParameters.size ∧
(∀i ∈ [1, f.inputParameters.size]:
 f.$inputParam_i$.type =▷$f^{\odot}$.$inputParameter_i$.type ∧
 f.$inputParam_i$.isOrdered =$f^{\odot}$.$inputParam_i$.isOrdered ∧
 f.$inputParam_i$.isMultivalued=$f^{\odot}$.$inputParam_i$.isMultivalued)

It may be noted that this is the same as:

f.inputParameters.size = $f^{\odot}$.inputParameters.size ∧
(∀i ∈ [1, f.inputParameters.size]:
 Typ_{Cnf} (f.$inputParam_i$, $f^{\odot}$.$inputParam_i$) ∧
 Mlt_{Cnf} (f.$inputParam_i$, $f^{\odot}$.$inputParam_i$))

Which is the formulation of Sig_{Cnf}. Therefore, if Sig_{Cnf} holds, (10) holds as well. QED.

Result_pins: "The number of *result* output pins must be the same as the number of output (*inout*, *out* and *return*) parameters of the called behaviour or operation. The type, ordering and multiplicity of each result Output pin must be consistent with the corresponding input Parameter". I.e., denoting the list of the templated action's output pins as *outPins*:

outPins.size = f.outputParameters.size ∧
(∀i ∈ [1, outPins.size]:
 f.$outputParam_i$.type =▷ $outPin_i$.type ∧
 f.$outputParam_i$.isOrdered = $outPin_i$.isOrdered ∧
 f.$outputParam_i$.isMultivalued = $outPin_i$.isMultivalued)

It must be proved that:

outPins.size = $f^{\odot}$.outputParameters.size ∧
(∀i ∈ [1, outPins.size]:
 $f^{\odot}$.$outputParam_i$.type =▷ $outPin_i$.type ∧
 $f^{\odot}$.$outputParam_i$.isOrdered = $outPin_i$.isOrdered ∧
 $f^{\odot}$.$outputParam_i$.isMultivalued = $outPin_i$.isMultivalued)

The demonstration of this formula is similar to the previous one, with output pins and parameters instead of input ones and swapping the left-hand side with the right-hand side of all comparisons. QED.

4.2.6 On Operation Call Actions

These are actions that specifically invoke operations.

Type_target_pin: "(…) the operation must be an owned or inherited feature of the type of the *target* input pin (…)" (OMG 2015, sec.16.14.10.6). In *operation call* actions, the pin that receives the executing instance is named "target" in the UML metamodel. In the archetypal action, it is represented by the *self* pin. Therefore, this constraint has the same formulation and demonstration as *Object_type*, in section 4.2.2.

4.2.7 On Calls to Operations on Collections

This section focus on a topic about which (OMG 2015) is either ambiguous or silent: the call of operations on collections. Expressions such as 'customers.count()' should be representable as in Figure 20.

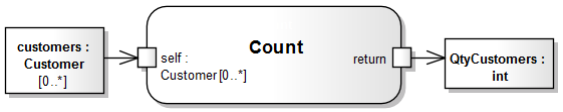

Figure 20: Collection operation call.

However, it could be noted that the action in Figure 20 seems to violate constraint *Type_target_pin* in the section 4.2.6, since *Count()* is not a feature of *Customer*. This is not a definite violation because, in fact, (OMG 2015) is not clear in what is meant by "the type of the 'target' [*self*] pin" in the aforementioned constraint when that pin is multivalued. If such type is interpreted as a collection type – i.e., in Figure 20 "the type of *self*" would be *Set{Customer}* – action in Figure 20 complies with the constraint. Contrarily, if (OMG 2015) refers to the type of the individual objects passed through the multivalued pin, then the action in Figure 20 violates the constraint. Furthermore, the latter case would mean that UML's *call operation action* is not intended to represent calls to operations on collections. But then, there are no other kind of action in UML that could represent such calls. Consequently, in this paper we make it clear what would be the counterpart of constraint *Type_target_pin* when the *self* pin is multivalued, declaring a constraint additional to those in (OMG 2015):

- *Call_of_collection_operation*: If the *self* pin is multivalued, then the type that owns *f* must be a collection; and vice versa. I.e.: self.isMultivalued ⇔ (f.owner =▷ Collection) (P10).

It must be proved that: caller$^{\odot}$.isMultivalued ⇔ (f$^{\odot}$.owner =▷ Collection) (11). This is ensured by Mlt_{Cnf} and $STyp_{cnf}$: (P10), Mlt_{Cnf} (caller, caller$^{\odot}$), $STyp_{Cnf}$ (f.owner, f$^{\odot}$.owner) ⊢ (11). QED.

4.2.8 On 'Create Object' Actions

In this kind of actions (see first two examples in Figure 6) *f* is a classifier. Thus, *f* is renamed to *C*, for clarity reasons.

- *Classifier_not_abstract*: "The classifier cannot be abstract" (OMG 2015, sec.16.14.18.5). I.e.: ¬ C.isAbstract (P11).

Abstraction Conformance proves that the same holds in the target space: (P11), $Abst_{Cnf}$ (C, C$^{\odot}$) ⊢ (¬ C$^{\odot}$.isAbstract). QED.

- *Multiplicity*: "The multiplicity of the *result* output pin is 1..1" (OMG 2015, sec.16.14.18.5). In terms of the action template: ¬ result.isMultivalued (P12).

Mlt_{Cnf} proves that the same holds in the target space: (P12), Mlt_{Cnf} (result, result$^{\odot}$) ⊢ (¬ result$^{\odot}$.isMultivalued). QED.

- *Same_type*: "The type of the *result* output pin must be the same as the classifier" (OMG 2015, sec.16.14.18.5). This constraint is always true, because in programing languages, 'new MyClass' always yields an object of *MyClass*.

The constraint *Classifier_not_association_class* is not applicable because the concept of Association Class does not exist in the common OOP languages.

5 RELATED WORK

Research aiming at improving the UML Template model is scarce. (Caron & Carré 2004) and (Vanwormhoudt et al. 2015) are the pieces of work most affine to the this paper.

Like the current paper, (Caron & Carré 2004) also proposes a set of rules, additional to that of UML, as a mean to enforce the well-formedness of elements bound to templates. (Caron & Carré 2004) is not clear on the purpose of the well-formedness it is trying to achieve. If it were compilability assurance, it overlooks several important aspects that are referred in the current paper, such as multiplicity, staticity, and visibility. FC take such aspects under consideration.

(Vanwormhoudt et al. 2015) proposes an extension to the UML Template concept called Aspectual Template (AT). Instead of having multiple parameters exposing potentially disconnected elements, ATs have a single parameter, which exposes a model as a whole. Associated to ATs, a set of constraints ensures that the target model fragment is conformant with the AT parameter. However, the conformance being checked doesn't target the guarantee of compilability for the bound element. E.g., AT's constraints overlook multiplicities and the static nature of features, which are essential for compilability. Additionally, by not taking subtyping into consideration in some circumstances, AT's constraints are exaggeratedly strict. Our approach overcomes such limitations.

Although not strictly aimed at UML Templates, (France et al. 2004) proposes a technique for specifying Design Patterns and checking if such templates are applicable to their application model fragment. Its approach includes a concept termed *Role* that closely resembles the UML's Stereotype concept and a notation that allows superimposing Roles' metamodel constraints on model diagrams. Although the approach outpaces UML templates in

expressiveness, the conformance verification method overlooks several aspects essential to compilability, such as multiplicity and signature conformance.

Differently from the current paper, none of the aforementioned provides a formal proof of the effectiveness of their contributions.

In the Programing Languages field, both C++ Templates (Stroustrup 2013; Vandevoorde et al. 2002) and Java Generics (Gosling et al. 2014; Naftalin & Wadler 2006) check the adequacy of an actual argument to a template parameter taking into account the use that the template does of that parameter. This roughly results in the same level of validation as the one provided by FC. Whether non-conformities are accurately imputed to a bad argument-to-parameter assignment or badly reported elsewhere in the template instance, tends to be tool-specific. This may raise the argument that template instantiation verifications could be delegated to the target language, and FC be dismissed. However, there are plenty of reasons to perform such verifications on the UML model. The most evident lies is the fact that UML models may not be targeted to a language with generics. E.g., UML may be used to model databases and SQL doesn't possess generics. A not so obvious reason is that the FC's criteria may leverage computer assisted binding, with substitutes being automatic and semi-automatically elicited out of the target model.

6 CONCLUSIONS AND FUTURE WORK

Building a proof was undoubtedly useful, not only because it confirmed the theory put forth, but also because it unrevealed issues that otherwise might become unnoticed. These were mostly related with the substitution of classifiers and lead to the introduction of the *Subtyping* and *Abstraction Conformance* criteria. The former was not initially apparent because Typ_{Cnf} seemed to suffice for the purpose under consideration, being Subtyping Conformance just a corollary Typ_{Cnf}. The impossibility of using Typ_{Cnf} to prove the well-formedness of object flows forced to consider the hypothesis of Subtyping Conformance not really being a corollary of Typ_{Cnf}. That was evidenced only through a proof by contradiction (not shown in this paper). $Abst_{Cnf}$ was not detected previously because none of the empirically tested templates included a *new* statement that could be substituted by an abstract class. It looks like a formal proof is worth a thousand tests.

The demonstration strategy use only proves that FC is sufficient to ensure compilability. As a next step, a demonstration that shows that FC's rules are the necessary ones must be done.

REFERENCES

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

Farinha, J. & Ramos, P., 2015. Extending UML Templates towards Computability. In *MODELSWARD 2015, 3rd Int. Conf. on Model-Driven Engineering and Software Development*. ScitePress.

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

Gosling, J. et al., 2014. *The Java Language Specification, Java SE 8 Edition* 1st ed., Addison-Wesley.

Naftalin, M. & Wadler, P., 2006. *Java Generics and Collections*, O'Reilly Media, Inc.

OMG, 2015. *OMG Unified Modeling Language, version 2.5*, Available at: http://www.omg.org/spec/UML/2.5.

Stroustrup, B., 2013. *The C++ Programming Language* 4th ed., Addison-Wesley.

Vandevoorde, B.D., Josuttis, N.M. & Date, P., 2002. *C ++ Templates : The Complete Guide* 1st ed., Addison-Wesley.

Vanwormhoudt, G., Caron, O. & Carré, B., 2015. Aspectual templates in UML. *Software & Systems Modeling*.