Nested OSTRICH: Hatching Compositions of Low-code Templates

João Costa Seco
NOVA University Lisbon, NOVA LINCS
Caparica, Portugal
joao.seco@fct.unl.pt

Hugo Lourenço
OutSystems
Lisbon, Portugal
hugo.lourenco@outsystems.com

Joana Parreira
NOVA University Lisbon, NOVA LINCS
Caparica, Portugal
jb.parreira@campus.fct.unl.pt

Carla Ferreira
NOVA University Lisbon, NOVA LINCS
Caparica, Portugal
carla.ferreira@fct.unl.pt

ABSTRACT

Low-code frameworks strive to simplify and speed-up application development. Native support for the reuse and composition of parameterised coarse-grain components (templates) is essential to achieve these goals. OSTRICH — a rich template language for the OutSystems platform — was designed to simplify the use and creation of such templates. However, without a built-in composition mechanism, OSTRICH templates are hard to create and maintain.

This paper presents a template composition mechanism and its typing and instantiation algorithms for model-driven low-code development environments. We evolve OSTRICH to support nested templates and allow the instantiation (hatching) of templates in the definition of other templates. Thus, we observe a significant increase code reuse potential, leading to a safer evolution of applications. The present definition seamlessly extends the existing OutSystems metamodel with template constructs expressed by model annotations that maintain backward compatibility with the existing language toolchain. We present the metamodel, its annotations, and the corresponding validation and instantiation algorithms. In particular, we introduce a type-based validation procedure that ensures that using a template inside a template produces valid models.

The work is validated using the OSTRICH benchmark. Our prototype is an extension of the OutSystems IDE allowing the annotation of models and their use to produce new models. We also analyse which existing OutSystems sample screens templates can be improved by using and sharing nested templates.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Visual languages**; **Patterns**; *Frameworks.*

ACM Reference Format:
João Costa Seco, Hugo Lourenço, Joana Parreira, and Carla Ferreira. 2022. Nested OSTRICH: Hatching Compositions of Low-code Templates. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552442

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552442

1 INTRODUCTION

Abstraction and parametrisation are amongst the most significant mechanisms in programming languages to promote modularisation and code reuse [19]. They are present in programming languages from basic function definitions to sophisticated type systems [2] or type-level computations in metaprogramming mechanisms [3, 7].

The core contribution of this paper is to improve the template developer experience in low-code frameworks by allowing the reuse and composition of coarse-grain components. To that end, we equip the template language OSTRICH [20, 21] with a new built-in abstraction mechanism. Up to this point, OSTRICH allowed for the top-level reuse of template application components, integrated into the IDE functionality. We build on a library of components that encompasses templates with curated and tested functionalities, user interfaces, and business rules, to build applications in a faster and more reliable fashion. Without using a composition mechanism like the one introduced in this paper, full-fledged application templates would be cumbersome to create and impractical to maintain.

The template mechanism introduced in [20] covers more than 50% of template uses recorded by the platform. The time spent on choosing and adapting a template to a particular scenario is reduced from a few hours, for an experienced developer, to a few minutes, for all levels of expertise. Next, we introduced type dependencies between template parameters [21] in OSTRICH. This allows for more cases to be safely captured. Notably, we safely capture the case of a template that receives an entity and an attribute and requires that the attribute belongs to said entity. Some basic mechanisms of OSTRICH are nowadays at the foundations of the language in the OutSystems platform when instantiating sample screens. Namely, in the rendering of said screens with optional components. It is foreseeable that many of OSTRICH's features will inform future versions of the OutSystems language and platform. The state of the art in OSTRICH templates improves the life of a developer using templates. With OSTRICH, non-experts, low-code developers can reuse trustworthy predefined and parameterisable code. OSTRICH templates are developed by experts, ranging from professionally designed user interfaces to complex distributed systems algorithms.

OSTRICH templates follow an approach similar to other model-driven low-code programming approaches, like templates in UML (MOF) [23, 31, 32] and MetaDepth [11], where templates are instantiated in place by an external mechanism. In OSTRICH, even

though the instantiation algorithm in [20, 21] is embedded into the behaviour of the development environment, nevertheless, it is not a part of the semantics of the language. The main difference with model-driven approaches is that OSTRICH produces code at the same level of abstraction as the template definitions. In this paper, we present an extension of OSTRICH where the definition of a template can be made by the composition of other templates. Templates are still part of valid OutSystems application models that can be created, edited, and tested as normal application components. No other related template language has this characteristic. The default values for the parameters and all the sample application elements used as placeholders for a template instantiation are stubs created with the normal low-code toolchain (IDE and compiler). Our technical approach proceeds by extending the OSTRICH metamodel with a richer set of annotations that allow sample nodes to represent the composition of templates by instantiation of templates inside the definition of other (nested) templates.

Our developments are complementary to prior work [20, 21] in the sense that they improve the lives of template developers, and improve the quality of the template library. The previous instantiation mechanism, integrated IDE, was external to the language. Moreover, the validation of template instantiation (typing arguments against parameter specification) was also decoupled from the type system of the language. By defining a composition mechanism for templates we allow for an even more modular development and reasoning that reduces the effort of producing templates and increases the reuse potential of the language. The approach is central to the GOLEM project [15] where program synthesis is being used to generate component assemblies from high-level programming concepts. One goal of this project is to find alternative assemblies of larger components or full-blown applications that adapt to a considerable number of situations. Also, Machine Learning techniques could be used to identify common application patterns in the large corpus of OutSystems code and produce valid OSTRICH templates.

We present our model-driven approach via a running example that gets abstracted from a concrete application model of a screen to a reusable set of templates that can then be used to build many different applications. We also explain the instantiation and validation algorithm that is integrated into our prototype implementation. The validation of our proposal proceeds by using the same benchmark as [20], further abstracting the examples with nested templates and highlighting the reuse and sharing of smaller template user interface components. We give an account of the language impact in terms of reusability in the OutSystems' library of components.

Our contributions can be systematically presented as follows:

- A uniform composition mechanism for the OSTRICH template language. (Section 2)
- A backward compatible representation of the model-driven composition of templates. (Section 3)
- A one-pass instantiation algorithm that accounts for a wide variety of situations with cyclic dependencies between model elements. (Sections 4 and 5)
- A typing algorithm based on symbolic information that separates phases (compile and runtime) and accounts for the composition of templates. (Section 6)

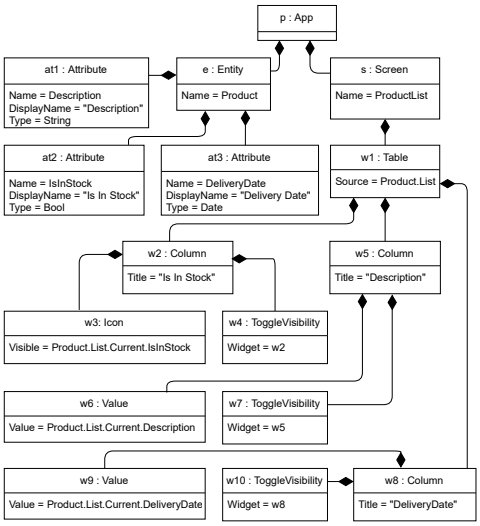

Figure 1: The instantiated application.

- An account for modularising templates in practice using an industry-standard benchmark. (Section 7)

Although this work may be interpreted as incremental, we highlight that it introduces a breakthrough in building, modifying, and viewing models. The presented results are significantly more generic, symbolic, and potentiate the reuse of curated code fragments.

2 NESTED TEMPLATES

In this section, we illustrate the concept of OSTRICH's nested templates by taking a new turn at modelling the application illustrated in [20], and extending and modularising it in a new way. We illustrate how nested templates can be reused and shared between different template definitions and therefore increase the productivity of software factories. Common templates, such as the ones in OSTRICH benchmark [20], contain styling options in user interface components, intricate algorithms, or code patterns that developers want to get right from start and keep uniform throughout an application. This promotes best practices on code reuse other than clone and own [12] on every single code pattern.

Figure 1 depicts the final stage of the application model in our example, which is an instance of the metamodel in Figure 2 that depicts a fragment of the low-code language of the OutSystems platform. OutSystems models follow a strict hierarchical structure where, for each object in the model, we identify the set of its *children* elements, available as a collection as named in the metamodel. Objects can also *use* other objects with no restrictions by using their names in expressions used to define the values for their attributes. For the sake of simplicity, in this paper, we support the definition of applications consisting of entities (database tables), screens and some selected widgets. We define screens as containing a tree of user interface widgets that can depend on entities to display data.

Our sample application consists of an entity named `Product` and a screen named `ProductList`. Entity `Product` has three attributes: `Description` of type `String`, `IsInStock` of type Bool, and `DeliveryDate` of type `Date`. Screen `ListProduct` contains a

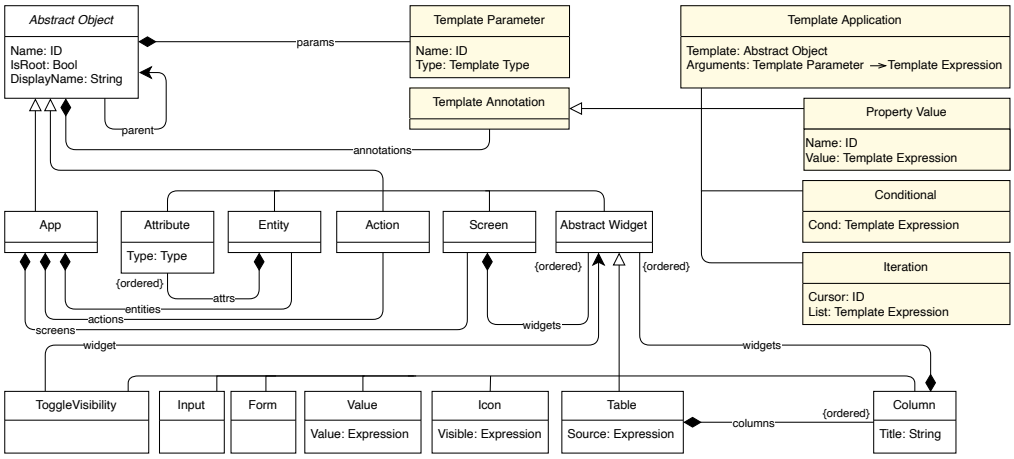

Figure 2: The OSTRICH metamodel.

widget `Table` with an explicit data dependency to entity `Product`, defined by the expression in attribute `Source`. The table contains three `Column` widgets. All columns include a `ToggleVisibility` widget that allows the end user to manually control the visibility of the corresponding column. The value of attributes `Description` and `DeliveryDate` are displayed using a generic `Value` widget. For the `IsInStock` attribute, we use a widget `Icon` whose visibility is determined by the attribute's value. The pattern we are capturing in this example, a screen used for listing the content of a database table, is common enough that we might want to abstract and isolate it into a reusable template, parameterized by the entity to be displayed. Such a template may, for instance, include an elaborate design that one wants to propagate uniformly throughout the application. Such a template can be modelled in `OSTRICH` as depicted in Figure 3, where annotations nodes are conservatively added (in yellow) to a regular model. The root node of the template is identified by property `IsRoot` set to `true`. The template nodes are green-coloured for readability purposes. Notice that we still need all other nodes in a model to make it a valid OutSystems application model but they are not part of the template. In the example of Figure 3, node `s1` is the root of the template; the `Template Parameter` annotations (`t1` and `t2`) declare parameters `e` of type `Entity` and `attr` of type list of attributes; the `Property Value` annotations (`t3` and `t4`) define the value of the related properties upon the instantiation of the enclosing template; an `Iteration` annotation (`t5`), defines that the referred node is replicated in this point of the model and instantiated for each one of the elements of the given list; and a `Template Application` annotation (`t6`) defines that said node will be replaced by the instantiated elements of the referred template. Said template, defining a column, defined in Figure 4. In that (nested) template, `Conditional` annotations (`t3` and `t5`) conditionally determine the widget to be used depending on the type of the attribute.

Notice that types for entities and attributes use a special name (`N`) to establish a relationship that needs to be preserved. In this case, the type system ensures that the list given as an argument includes only attributes of entity `e`. Such compile-time names are inferred from the definition of the templates. Fully integrating and checking such abstraction and instantiation mechanisms in the `OSTRICH` language and prototype is the core contribution of this paper. Note that template models can only be defined using a prototype of the OutSystems IDE with support for `OSTRICH` annotations.

Prior work [20] presents a parameterisable version of the model that instantiates this application given an entity as an argument to a declared parameter. We then extended it with the type system [21] that statically checks the dependencies between parameters referred to above. In this paper, we further increase the expressiveness of `OSTRICH` by allowing the instantiation of templates inside the definition of other templates, thus supporting the reuse and sharing of code between template definitions. One challenge is to conservatively introduce this extension, which means that `OSTRICH` applications, with instantiation nodes, are still compatible with the existing toolchain in the platform – compiler, editors, application builders, etc. The metamodel for `OSTRICH`, depicted in Figure 2, contains white coloured elements that correspond to the metamodel of OutSystems applications, and yellow coloured elements that correspond to annotations that conservatively extend the metamodel. We have a uniform approach to defining as a template any element with property `IsRoot` set to `true`, and the instantiation of templates as the application elements that contain the annotation `Template Application`. Importantly, the instantiation node is not required to be inside a template. Figure 5 illustrates an instantiation in the context of an application that triggers the instantiation algorithm.

Going back to the description of our running example, node `w2` in Figure 3 is repeatedly cloned, and each clone is replaced by the instantiation of a template `p2.w2`, with each one of the attributes as argument (and the entity). Template `p2.w2` is defined by the node `w2` in Figure 4 and its children (coloured in green). The depicted model is still a complete model, which makes it possible to define and verify the template in a context where some sample elements are used. Notice also that Figure 3 includes node `w3`, which is necessary to make the application model valid. This node is discarded entirely when node `w2`, a `Template Application`, is instantiated. The creation of the main application, in Figure 5, is a regular application model with an empty screen annotated with an `Template Application` annotation referring to template `p1.s1` (Figure 3).

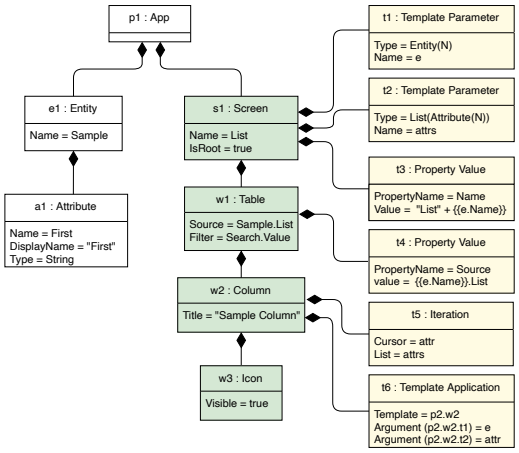

Figure 3: A template for tables using a template for columns.

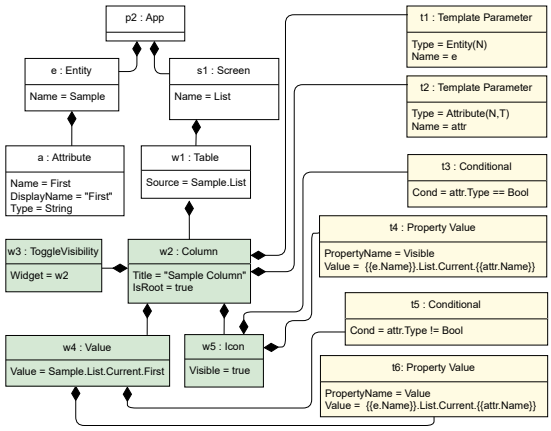

Figure 4: The column (inner) template.

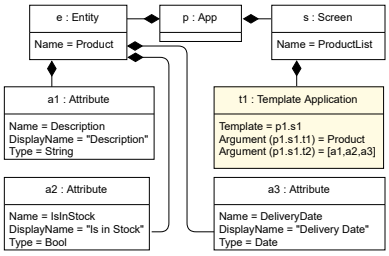

Figure 5: The creation of the main application.

The template preprocessor runs an instantiation algorithm and produces as a result the model in Figure 1. The definition of nested templates allows for the reuse of the "Column" template in other scenarios, such as the one in Figure 6 that also includes search functionality and another template instance of a form (`p7.w3`) to introduce new elements (elided from the paper for space reasons).

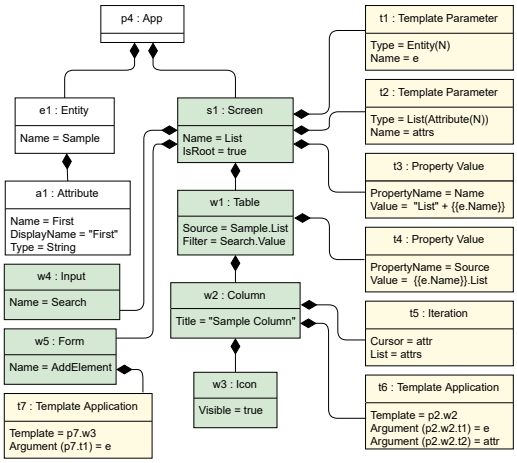

Figure 6: Column template being reused in a more sophisticated table template, with more (sub)templates.

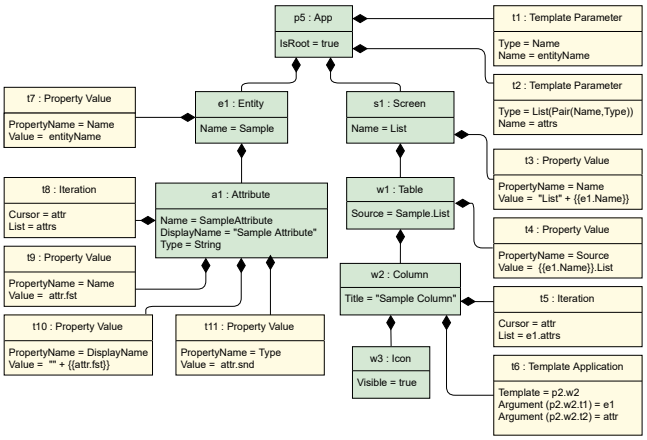

Figure 7: A template of a complete application.

The main challenges are related to representing standard mechanisms as the definition of abstractions and the instantiation of abstractions in a meta-model, representing those as sound model transformations while maintaining backward compatibility of the model in an industrial-grade tool. The algorithmic challenge lies in calculating the dependency graph between nodes and evaluating the transformations using a topological order so that the result of instantiating some node can be used in another node. This is visible in Figure 7 where the argument is a list of names that are used to create attributes and said attributes are later used to create columns. This means that a two-pass algorithm, one for creating nodes and another to assign values to properties, is no longer enough. Section 5 presents an algorithm that first computes a sound order of instantiation and then uses it to correctly instantiate all the nodes.

We end our template definition exercise with an example of a complete application defined by a template. In this case, the template (Figure 7) replaces all nodes in a caller application (Figure 8) with instances of the template nodes.

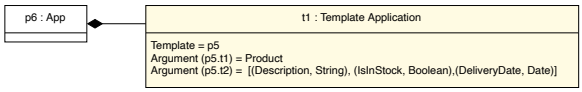

Figure 8: The creation of the main application (New).

3 THE TEMPLATE METAMODEL

We refine and extend the metamodel presented in [20] so that the application of a template can be uniformly used in the application models. The metamodel (Figure 2) defines the full language of annotations that can be added to nodes in a OutSystems model. Uncoloured elements correspond to (a simplified version of) the metamodel for OutSystems applications. The coloured elements are the ones that were introduced specifically by OSTRICH to support the definition of templates and template components.

Briefly, applications comprise multiple instances of `Abstract Object` nodes. These include entities (cf. database tables), entity attributes, computational actions (cf. function declarations), application screens, and user interface widgets. The original model describes more kinds of nodes which were omitted here for the sake of simplicity and space. Only the nodes used in the example were included in this metamodel. Nodes of kind `Abstract Object` may contain other nodes, thus forming a parent-child hierarchical tree structure like the one between entities and attributes. `Abstract Object` nodes can also use other nodes. For instance, widget `ToggleVisibility` uses its `Widget` property to refer to the widget whose visibility it is controlling. OSTRICH extends the OutSystems metamodel by adding the following new elements:

- Template Parameter annotation: declares a typed name to be used in the template's annotations.
- Template Application annotation: instantiates a template with given arguments and replaces the annotated node.
- Property Value annotation: dynamically defines at instantiation time the expression that establishes a property value.
- Iteration annotation: replaces an element with a collection of elements, once for each item in the provided list.
- Conditional annotation: dynamically includes or excludes an element, depending on the condition compile-time value.

In relation to [20] we specifically added one kind of annotation, the `Template Application` annotation, and extended template expressions to allow references to template elements.

The template-specific metamodel elements are treated as annotations on the base metamodel. This allows us to maintain backward compatibility with existing tools, which can just ignore the annotations, and at the same time facilitates extending the tools that need to take advantage of the annotations.

There are well-formedness constraints that are captured in the metamodel directly and not in the type system that focuses on constructing well-typed expressions in value properties. Such constraints limit the kinds of nodes that can be used as children of a node. For example, a widget `Table` contains only widgets `Column`, and a widget `Column` may contain any kind of abstract widget. Other rules, related to annotations, need to be explicitly checked on the model. Examples of rules are the absence of template definitions inside other templates or the use of recursion in templates.

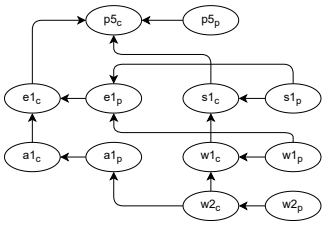

Figure 9: Dependency graph for the template of Figure 7.

4 DEPENDENCY ANALYSIS

During the instantiation process, each node in the template is subjected to two operations: one to create the necessary objects in the target app, and the other to set the properties of those objects. The order by which operations are carried out is relevant due to possible dependencies between template nodes. For instance, in Figure 7 the name of Screen `s1` is determined by the template expression `"List"+{{e1.Name}}` in Property Annotation `t3`. This template expression refers to the name of Entity `e1`, and thus can only be evaluated after the properties of `e1` have been defined.

To determine a valid creation order, we build a dependency graph that captures the relationships between the instantiation operations. For each relevant template node[1] two nodes are created in the dependency graph, with subscripts c and p to denote the `CreateObjects` and `SetProperties` operations, respectively. Dependencies between nodes are established according to the rules presented below. Figure 9 depicts the dependency graph for the template of Figure 7 to help illustrate these rules.

(1) Objects created before initializing properties, e.g. $p5_c \leftarrow p5_p$.
(2) Parents created before their children, e.g. $p5_c \leftarrow e1_c$.
(3) For child collections where the order is relevant, siblings' order must be preserved, e.g. $w4_c \leftarrow w5_c$ for the template of Figure 6.
(4) Nodes whose template expressions in annotations refer to other nodes created after the properties for referenced nodes have been set, e.g. $a1_p \leftarrow w2_c$. Due to template expression `e1.attrs`, $w2$ depends explicitly on $e1$ and implicitly, by transitivity, on its attributes. Note that we want to evaluate `e1.attrs` to the list of the attributes of the newly created entity `e1`, thus requiring the attributes to have been fully processed by the time we start processing `w2`.
(5) Nodes with `Property Value` annotations whose template expressions reference other nodes must have their properties set after the properties of the referenced nodes have been set, e.g. $e1_p \leftarrow s1_p$, with `e1` used in `"List"+{{e1.Name}}`.

The order by which operations are carried out is determined by a topological order of the dependency graph. If the graph contains loops and thus no topological order can be established, then the template is deemed invalid.

The previous version of OSTRICH [20] used a two-pass algorithm that traversed twice the template node tree – first to create all objects and then to evaluate their properties. A two-pass approach only allows for model-level dependencies. An example of such a

[1]Child nodes of nodes containing a `Template Application` are sample models and are ignored. That is the case of node `w3`.

dependency can be found in Figure 4, where widget `w3` references widget `w2` via its `Widget` property. The dependency analysis introduced here allows for more expressive templates in which dependencies can also occur via template expressions, which may refer to objects that will be created when the template is instantiated. That is the case of `Property Value` annotation `t3` in Figure 7.

5 INSTANTIATION ALGORITHM

Every application model, with or without annotated nodes, is pre-processed when first *published*[2] in the platform. Templates are expanded in the current model and can be incrementally changed after their publication. For instance, when pre-processing the model in Figure 5, the algorithm finds the `Template Application` annotation `t1` associated to screen `s`. The corresponding table template (`p1.s1` in Figure 3), whose root is a screen element, is instantiated in the target model to replace screen `s` and using the local entity declaration `Product` and its attributes as argument. Since the model of a template is a valid OutSystems model, pre-processing a template definition model, with template declaring annotations, results in the application model using the default values and ignoring said annotations. The instantiation algorithm recursively replaces nodes annotated with top-level `Template Application` annotations by cloned and instantiated versions of the template model referred by said annotations. The instantiated annotation `Template Application` is linked to the resulting root node so that it can be refreshed and reapplied if needed. However, reapplying a template overwrites any changes performed in the generated code.

We illustrate the instantiation algorithm (Algorithm 1) using a template (Figure 7) and a target app (Figure 8) and the inputs in the `Template Application` annotation `t1` in Figure 8:

- `template` = `p5` (template of Figure 7)
- `targetParent` = `null` (annotation `t1` is applied to an app, which doesn't have a parent object)
- `args` = { `entityName` ↦ "Product", `attrs` ↦ [("Description", String), ...] }

The algorithm starts by calculating the dependency graph for the template (Figure 9) and uses a topological order of the graph as the sequence of operations to carry out (Line 2). The root evaluation environment, `rootEnv`, is initialized with the arguments (`entityName` and `attrs`). The map `newObjs` keeps the objects that are incrementally created by the instantiation algorithm. The map key is a template node and the value is a map associating an evaluation environment with the object created from the template node using that environment. This representation allows us to get detailed information about all the objects that have been created for a given template node (Lines 7 and 11), and also pinpoint the specific instance created for a particular environment (Lines 45 to 48). To bootstrap the algorithm, we initialize `newObjs` with a value that maps the template's parent node to `rootEnv` and `targetParent` (Line 4). In our example both the template's parent node and `targetParent` are `null`. Figure 10 depicts the algorithm initial state, and Figure 11 the state after processing operations $p5_c$, $e1_c$, $a1_c$, and $a1_p$.

The algorithm then proceeds to carry out the operations already prepared. The `CreateObjects` operations start by fetching all the evaluation environments and objects for the template node's parent (Line 7). For the first operation, $p5_c$, this corresponds to the initial and single value in `newObjs`, which is {`rootEnv` ↦ `null`}. `Conditional` annotations (Lines 15 to 17) are processed as expected:

[2]In the OutSystems platform publishing an application corresponds to, in a single step, generating code and deploying the application to a cloud-based infrastructure.

Algorithm 1 Template instantiation algorithm

```
    types
        Operation = AbstractObject × { CreateObjects, SetProperties }
        Env = (ID ⟶ Object) × Env
                ▷ evaluation environment (with reference to parent environment)
        NewObjs: AbstractObject ⟶ (Env ⟶ AbstractObject)
                ▷ map of new objects indexed by template node and environment
    input
        template: AbstractObject          ▷ root template object with annotations
        targetParent: AbstractObject                    ▷ target parent object
        args: TemplateParameter ⟶ Object                ▷ template arguments
    locals
        operations: Sequence of Operation               ▷ instantiation operations
        rootEnv: Env                                    ▷ root evaluation environment
        newObjs: NewObjs
 1: function INSTANTIATE(template, targetParent, args)
 2:     operations ⟵ TOPOLOGICALORDER(template)
 3:     rootEnv ⟵ newEnv(args)
 4:     newObjs ⟵ { getParent(template) ↦ { rootEnv ↦ targetParent } }
 5:     for all (templNode, op) in operations do
 6:         if op = CreateObjects then
 7:             parentsOfNode ⟵ get(newObjs, getParent(templNode))
 8:             for all (env ↦ parent) in parentsOfNode do
 9:                 CREATEOBJECTS(templNode, parent, env, newObjs)
10:         else
11:             objsInNode ⟵ get(newObjs, templNode)
12:             for all (env ↦ newObj) in objsInNode do
13:                 SETPROPERTIES(templNode, newObj, env, newObjs)
    input
        templNode: AbstractObject                       ▷ current template object
        targetParent: AbstractObject                    ▷ current target parent object
14: function CREATEOBJECTS(templNode, targetParent, env, newObjs)
15:     if hasConditionalAnnotation(templNode) then
16:         if evaluate(getCondExpression(templNode), env) == true then
17:             CREATEOBJECT(templNode, targetParent, env, newObjs)
18:     else if hasIterationAnnotation(templNode) then
19:         list ⟵ evaluate(getListExpression(templNode), env)
20:         cursorName ⟵ getCursor(templNode)
21:         for all item in list do
22:             newEnv ⟵ beginScope(env)
23:             bind(newEnv, cursorName, item)
24:             CREATEOBJECT(templNode, targetParent, newEnv, newObjs)
25:     else CREATEOBJECT(templNode, targetParent, env, newObjs)
26: function CREATEOBJECT(templNode, targetParent, env, newObjs)
27:     if hasTemplateApplicationAnnotation(templNode) then
28:         template ⟵ getTemplate(templNode)
29:         args ⟵ map(getArguments(templNode),
                (param, expr) ⟶ (param, evaluate(expr, env)))
30:         INSTANTIATE(template, targetParent, args)
31:     else
32:         newObj ⟵ createChild(targetParent, typeof templNode)
33:         bind(env, getHandle(templNode), newObj)
34:         objsInNode ⟵ get(newObjs, templNode) ∪ { env ↦ newObj }
35:         newObjs ⟵ newObjs ∪ { templNode ↦ objsInNode }
36: function SETPROPERTIES(templNode, newObj, env, newObjs)
37:     for all prop in getProperties(templNode) do
38:         value ⟵ evaluateProperty(templNode, prop, env, newObjs)
39:         setPropertyValue(newObj, prop, value)
    input
        prop: Property                                  ▷ property to be evaluated
40: function EVALUATEPROPERTY(templNode, prop, env, newObjs)
41:     if hasPropertyAnnotation(templNode, prop) then
42:         return evaluate(getValueExpression(templNode, prop), env)
43:     else
44:         value ⟵ getPropertyValue(templNode, prop)
45:         if contains(newObjs, value) then        ▷ value is a template object
46:             objsForValue ⟵ get(newObjs, value)
47:             if contains(objsForValue, env) then
48:                 value ⟵ get(objsForValue, env)
49:         return value
```

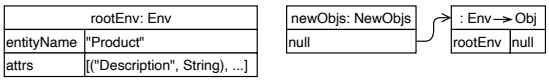

Figure 10: Initial state of the algorithm.

if their condition evaluates to `true` then the template node is processed, otherwise it is ignored. For `Iteration` annotations (Lines 18 to 24) for each element in the evaluated `list` we create a new child evaluation environment. The template node is then processed using the new environment. If neither `Conditional` nor `Iteration` annotations are found then the template node is processed normally (Line 25). `Template Application` annotations (Lines 27 to 30) are processed by recursively calling the `Instantiate` function. The creation of new objects occurs in Lines 32 to 35. Notice that we store objects together with the evaluation environment in `newObjs`.

For `SetProperties` operation the process is straightforward. We fetch all the evaluation environments and objects for the template node (Line 11). For operation $a1_p$, this corresponds to {`env1` $\mapsto$ `at1`, `env2` $\mapsto$ `at2`, `env3` $\mapsto$ `at3`}. Notice that we have three environments/objects to process, which are the result of evaluating Iteration annotation `t8` while processing operation $a1_c$. Function `SetProperties` function is called for each element of `objsInNode`. Each property of the new objects is set either using a Property annotation (if found) or by copying the value from the template node. Consider, as an example, the `env1` $\mapsto$ `at1` case and property `Name`. The template expression `attr.fst` (Property annotation `t9` in Figure 7) evaluates do `Description` in environment `env1`, and thus the `Name` of `at1` is set to `Description`. A special and important case is that of properties whose value is a model object, which must be mapped to the correct object in the target app (Lines 45 to 48).

Newly created objects are stored in the evaluation environments (Line 33) so that we can properly evaluate template expressions that refer to template nodes. For instance, the template expression `e1.attrs` in `Iteration` annotation `t5` (Figure 7) refers to the (sample) entity `e1`. When processing this expression `e1` is evaluated, as desired, to entity `e`, according to `rootEnv`.

The algorithm presented here represents a significant evolution with relation to [20] where template instantiation is external to the language, like in traditional model transformation processes [11].

6 CHECKING MODEL SOUNDNESS

In addition to the instantiation algorithm embedded in the preprocessing phase of the publication process in Service Studio, we define a verification that checks the pre-conditions for the instantiation algorithm to produce well-formed runtime expressions. The use of a model-driven development environment ensures that all models are structurally valid by construction concerning the rules embedded in the metamodel. The introduction of orthogonal model-defining mechanisms, such as the use of templates, requires new validation methods. The first validation is that, when accounting for all dependencies introduced by template annotations, the model contains a topological order of nodes. The remaining validation is twofold. On the caller side, one needs to check the compatibility of the arguments used on each `Template Application` node against the corresponding interface. On the callee side, one needs to check the template definition against the specification of each parameter. We further explore this topic in Section 6.1.

We introduce the explicit validation of the application models which are produced by templates in terms of the node nesting rules that are allowed in the metamodel. The immediately visible rules in Figure 2 are: an `App` can only contain nodes of type `Screen`, `Action` or `Entity`; an `Entity` can only contain nodes of type `Attribute`; nodes of type `Screen` may contain any kind of widgets (i.e. nodes of type `Abstract Widget`); nodes of type `Table` can only contain nodes of type `Column`, which in turn may contain any kind of widget. Similarly, typechecking is performed in all template expressions defining the value of node properties, using the types of nodes and their properties. To enforce the discipline defined by the metamodel its categories are assigned to node types and include the rules in the type system. Crucially, in the case of conditional nodes and iteration nodes, multiple node types can be produced by a single annotation. Consider the model fragment in Figure 12, the node type for the children nodes of `w2` is `NodeType(ToggleVisibility,Value,Icon)`, and the set of common properties is the one at `Abstract Widget` level. Notice in Figure 2 that `ToggleVisibility`, `Value`, `Icon` do not have any local properties in common.

We assign types to nodes when defining the semantics of template expressions. Namely, when defining model nodes as parameters to templates. Each model node has a type. Entity nodes have type `Entity(N)`, with `N` being a compile-time name, or a compile-time (type) variable. Attribute nodes have type `Attribute(N,T)` where `N` is the name of the entity to which it belongs, and `T` is the actual type (or type variable) of the attribute value it represents. For instance, if one refers to entity `Product` with an attribute `Description` in an expression (Figure 7), it is represented by `Entity(Product)`, and its attribute by type `Attribute(Product, String)`. When dealing with records of attributes from an entity named `N`, we use type `RecordAttr(N)`. The label of an attribute of such entity is of type `LabelAttr(N,T)`, where `N` is the name of the entity, and `T` the type of the values it represents. We then extend this language of types with standard types. Records of elements have type $\{L_i : T_i{}^{i \in 1..n}\}$, where each label L_i maps to a type T_i. A label L has type `Label(`L`)`. Note that we omit types when convenient.

Besides defining new nodes of a target application model, templates also define new runtime expressions that define values of properties in the newly created nodes. To build well-formed expressions referring to model elements, we require extra information about the names used, namely parameters. We introduced [21] a limited form of dependency between parameters through their types. The particular case that is interesting to capture is the relation between entities or entities and their attributes via a compile-time name. We describe this mechanism in Section 6.2. Finally, in this paper, we separate the runtime part of expressions and the compile-time parts with a type discipline using the special type `Box(T)`, inspired by [10]. Although the source of our prototype cannot be made public for intellectual property reasons, we define and make public a textual language and reference type checking algorithm for the model presented here[3]. We present a series of examples as accessory materials in the repository.

[3]Please refer to the link https://github.com/jbp182/OSTRICH-OCaml

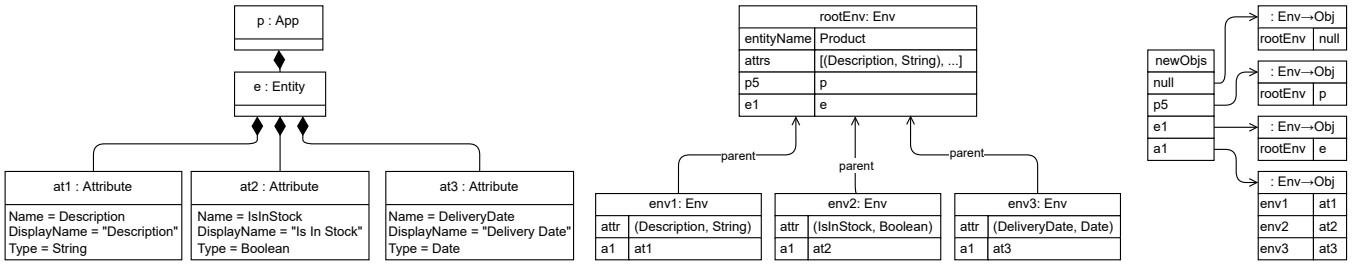

Figure 11: Algorithm state after executing operations $\mathtt{p5}_c$, $\mathtt{e1}_c$, $\mathtt{a1}_c$, and $\mathtt{a1}_p$.

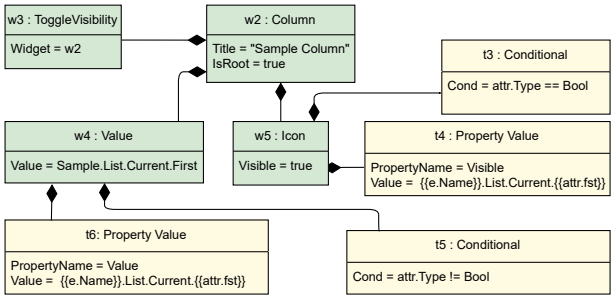

Figure 12: A node with alternative child node types.

6.1 Typing Template Applications

Since arguments used in `Template Application` nodes may not be actual model instances yet, we use symbolic information to check compatibility between arguments and parameters. Since types of parameters may have unbound names, we use unification to match types. Unification solves equations between symbolic expressions, i.e., finds a substitution for type variables under which two terms match [22]. Take the example of the template application in Figure 3 that instantiates the template `p2.w2` in Figure 4. The inner template (Figure 4) is parametrised by the compile-time values with the entity type `Entity(N)` and the attribute type `Attribute(N,T)`. In Figure 3, `e` and `attr` are the arguments used to instantiate the inner template. The type of `e`, `Entity(N')`, is defined in the parameter annotation `p1.s1.t1`. Since `attr` is an element from `attrs` (`p1.w2.t5`), and `attrs` is a list of attributes of entity `e` (`p1.s1.t2`), then `attr` has type `Attribute(N',Top)`. Here, `Top` represents any arbitrary attribute type, because `attr` is not an actual model instance, thus we do not know its values' actual type yet. Since `Top` behaves as a wildcard before instantiation, it was omitted from the example in Figure 3. This approach is sound due to the immutability of the attribute lists. Additionally, note that, in Figure 3 and Figure 4, the compile-time name `N` appears with the same syntax, despite being potentially different. Inside the same template definition, a name `N` preserves its connotation. However, this is not transversal between templates, i.e., templates can be instantiated with `N` referring to distinct compile-time names. Hence, we use names `N'` and `N` to distinguish between in different contexts. By unifying argument and parameter types (`N`, `N'`, `T`, and `Top`) we obtain the substitution $\mathtt{N} \mapsto \mathtt{N'}$ and $\mathtt{T} \mapsto \mathtt{Top}$, which is a valid substitution and ensures the correctness of the template application in `p1.w2.t6` (Figure 3).

In the (formal) functional implementation of the algorithm, we use universal quantification to declare such names and types rather than unification. In a low-code setting, we do not expect developers to explicitly specify type variables. In the syntactically controlled environment of the IDE prototype, the experience of declaring new names is yet to be designed. We use the implicit declaration of such variables and unification to compare them in the case of template application nodes. In the example above, the name `N` is used to denote a dependency between types, which we explore next.

6.2 Type dependencies

Name (`N`), used in parameters `p1.s1.t1` and `p1.s1.t2` in Figure 3 is used to define the types of an entity and a list of attributes. Recall that those names are, implicitly, universally quantified. Quantified names are unique, opaque, compile-time values that can be used to compare parameter types, link, or distinguish one from one another. For instance, we can detect if two parameters of two given entity types are aliases or if an attribute type parameter is linked to another attribute of an entity type. This is important when typing template expressions that build runtime expressions that must be (type) valid in the supporting model.

Template definitions contain nodes with mixed compile-time and runtime expressions. We define a staged computation strategy [10] when verifying and evaluating expressions to produce valid expressions and avoid harmful dependencies between compile-time and runtime expressions (phase errors) [4]. The algorithm depicted in Algorithm 2 detects phase errors using a runtime type environment, *r-env*, and a compile-time type environment, *c-env*. We restrict the typing of runtime expressions so that they only enclose other runtime expressions and variables from *r-env*.

Consider the example in Figure 4, namely the runtime expression of the `Value` property in `p2.w5.t4` and `p2.w4.t6`:

```
{{e.Name}}.List.Current.{{attr.Name}}
```

In this case, we use the syntax with double curly braces [20] to identify compile-time expressions embedded into runtime expressions. Notice that variables `e` and `attr` are compile-time variables. When instantiated in compile-time with entity `Product` and its attribute `Description` this expression evaluates to the runtime expression `Product.List.Current.Description`, that may be later evaluated to an actual string value. Both `e.Name` and `e.Label` are compile-time expressions, that evaluate to compile-time names, that are runtime expressions, `Product` and `Description`, respectively.

In this paper, we delimit compile-time expressions using the special concrete syntax `{{E}}`. In the functional implementation

Algorithm 2 Typechecking algorithm (partial)

```
   input
      expression: Term                          ▷ term expression to be typed
      c-env: Env                                ▷ compile-time environment
      r-env: Env                                ▷ runtime environment
 1: function typeOf(expression, c-env, r-env)
 2:    match expression with
 3:       x | x: T ∈ c-env ≜ T
 4:       u | u: T ∈ r-env ≜ Box(T)
 5:       M.Name | typeOf(M, c-env, r-env) = Entity(N) ≜
 6:          Box( {List: {Current: RecordAttr(N)}})
 7:       M.Name | typeOf(M, c-env, r-env) = Attribute(N, T) ≜
 8:          Box( LabelAttr(N, T) )
 9:       M1 . M2 | typeOf(M1, c-env, r-env) = {Li: Ti  i∈1..n}
10:             and typeOf(M2, c-env, r-env) = Label(Lj  j∈1..m)
11:             and Lj  j∈1..m ⊆ Li  i∈1..n  ≜ Tj
12:       M1 . M2 | typeOf(M1, c-env, r-env) = RecordAttr(N)
13:             and typeOf(M2, c-env, r-env) = LabelAttr(N′, T)
14:             and N = N′ ≜ T
15:       M2[{{M1}}] | typeOf(M1, c-env, r-env) = Box(T1)
16:             and typeOf(M2[u], empty, r-env ∪{u: T1}) = T2 ≜ Box(T2)
17:    end
```

of the algorithm, inspired by [10], we use the **box** constructor and the **let box** destructor. The two representations are isomorphic, where a runtime expression is enclosed by a **box** constructor and the use of the double curly braces corresponds to the use of a **let box** destructor for each compile-time subexpression.

Algorithm 2 shows a fragment of the typechecking algorithm limited to the expressions necessary for this example. We use a context-based syntax $E[M]$ to denote an expression E with an inner compile-time expression M. For instance, the expression above can be expressed using contexts, isolating compile-time subexpressions, by $(\square.List.Current.\{\{attr.Name\}\})[e.Name]$. Notice the typing of a runtime expression (M_2) assembled with another subexpression (M_1) in Line 15. The first guard guarantees that M_1 is a runtime expression, i.e., M_1 has type $\texttt{Box}(T_1)$. We guarantee that the remainder of the M_2 expression contains only runtime variables by typing M_2 with an empty compile-time type environment (Line 16). The result is a runtime expression with type $\texttt{Box}(T_2)$. By replacing the compile-time expression M_1 with an identifier, and looking at our example above we can isolate the second compile-time sub-expression $(u.List.Current.\square)[attr.Name]$ and proceed with the typing algorithm.

Within an entity, only its attributes are accessible, and therefore, in the aforesaid expression, `attr` must be an attribute of `e` for the expression to be well-typed. We ensure it through: the entity and attribute types, which contain a common name `N`; the resulting types of the expression `M.Name` (Lines 5 to 8), and the selection operation type represented as "**.**" (Lines 9 to 14). Any attempt to instantiate the model with an entity and an attribute of a different entity would not satisfy the guard $N = N'$ (Line 14), and the typechecking algorithm would reject the model instantiation annotation. These dependencies between types of parameters allow the definition of more diverse templates, by introducing restrictions to their applications and guaranteeing their appropriate instantiation and the production of valid models.

7 EVALUATION

We evaluated the new version of `OSTRICH` by looking for shared patterns in existing OutSystems screen templates. We specifically looked at the top 10 (out of 70) most used templates [20]. Collectively, this set of templates accounts for more than 50% of all screen template instantiations after four years of generalized use in the platform. A total of 7 shared patterns were discovered (Table 1) and represented as nested `OSTRICH` templates. The existing screen templates were successfully modified to take advantage of the new nested templates (Figure 13; existing screen templates are on the left, and the new nested templates are on the right and highlighted). Finding 7 shared patterns in a small sample of 10 screen templates is quite significant, and demonstrates that reuse occurs in practice. Notice the case of nested template `Attribute`, which is either directly or indirectly used by all (top-level) screen templates. This template strongly contributes to reducing complexity, since it allows to keep in a single place the knowledge and rules about how to properly visualize an entity attribute based on its type. This is best understood by referring back to Figure 4, which is a simplified variant of the `Atttribute` template and in which only the boolean type is handled. In reality, OutSystems has a total of 12 different basic types with distinct visualization rules. The addition of the 7 shared patterns results in an increase in our library of templates from 10 to 17 (a 70% increase). We have analyzed a small subset of screen templates (the top 10 out of 70), but we anticipate that more shared patterns will be found in the remaining screen templates.

Nested template	Description
Labelled attribute	Entity attribute with a text label
Pie chart	Aggregate data and display as a pie chart
Listing	Database data in a list format
Table	Database data in a table format
Attribute	Chooses widget for an entity attribute based on its data type
Filter	Applies a filter to a data source
Pagination	Page-based navigation in a large set of data

Table 1: Nested templates.

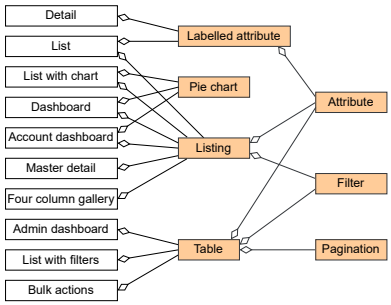

Figure 13: Screen templates using the new nested templates.

The introduction of nested templates in `OSTRICH` enabled the creation of a set of reusable building blocks that the complexity of existing templates and make it easier and faster to create new templates. With nested templates, it is feasible to produce templates that create full-fledged applications, as illustrated by Figure 7.

8 RELATED WORK

Model driven engineering. As stated in [27], low-code development is closely related to model-driven engineering, but low-code

principles, practices, and techniques have relevant differences from the ones of model-driven engineering. There are, however, intersection points between these two approaches. In particular, templating as proposed here can be seen as a Model-to-Model transformation (M2M) [9], since our template annotations describe the *transformations* to be applied to an OutSystems model. However, we argue that there are features in OSTRICH that could not be fully addressed with M2M. Namely, OSTRICH provides an integrated semantics of compile and runtime programming constructs, which is possible because templating is a seamless abstraction mechanism layered on top of the OutSystems metamodel. In contrast, M2M approaches as EMF [29] or MPS [17, 26] define transformations as external to the language itself (cf. ATL [1]). Moreover, the soundness guarantees provided the typechecking algorithm cannot be matched by M2M syntactic checks and semantic constraints expressed in OCL.

Multistage programming. In this approach, a program is divided into different levels of evaluation, available to the programmer through syntactic operators called staging annotations [30]. To support the algorithmic construction of programs at compile time, multistage programming has been for several mainstream functional programming languages, notably MetaML [30], MetaOCaml [18], and Template Haskell [28]. Our approach is inspired by richer type-level computations that reason about the structure of types and produce custom code constructions [3, 7]. The innovation of our work is the integration of multistage programming with type-level computations, in a low-code context. Crucially, we developed a typing algorithm to analyse nested template code guaranteeing that a well-typed instantiation produces well-typed code.

MetaDepth [11] is related to our approach to defining a DSL with a "generic" layer supporting nested templates. This layer allows the instantiation of high-level concepts at a lower level in the chain of models. OSTRICH constructs show/hide model elements and iterate over (compile-time) lists of (compile-time) values. These can also be expressed in MetaDepth through generators between different level models. The distinguishing feature between OSTRICH and MetaDepth is the verification of model conformance. In the latter, and other UML approaches [14, 31], model conformance is performed on the instances after parameter substitution. OSTRICH checks conformance statically by verifying the template and its arguments at compile-time.

General purpose programming languages. Most mainstream programming languages have some support for metaprogramming. From the basic level of lexical macros, like the ones supported by C, to bounded polymorphism, like Java generics [2]. The latter is closely related to parametric polymorphism [5] which abstracts the nature of the processed elements and does not take advantage of the structure of their arguments. As such, the concrete type or compile-time values that are used as parameters have minimal impact on behaviour customisation in the instantiated code.

UML Templates. Templates in UML [24] address model reuse through the concepts of abstraction and parametrisation [19], with some variants proposed and instantiated in EMF-based tools [6, 31, 32]. UML templating allows for the substitution of parameters and cloning model elements to produce other diagrams. In comparison, OSTRICH includes a full-fledged template language with constructs for nested templates, iteration, and conditional annotations, supported by a strongly-typed approach that provides safety properties. Moreover, we have defined and implemented a prototype for the formal semantics of the staged template expression language that represents OSTRICH. Similar verification results can be obtained using OCL [14], like in [32], or using contracts [8]. However, in both approaches [8, 32], it is not clear how to verify, at compile-time, the instantiation of model elements and the expressions being produced for the model instance. As with UML, OSTRICH supports the partial instantiation of parameters as it takes a conservative extension of the template base model where all parameters have default values. Unlike UML templates, our templates are models that can be viewed, edited, and compiled by the platform. This also accounts for a seamless evolution of the existing tool ecosystem.

Template Languages for Web interfaces. Textual template languages have long enabled the creation of dynamic pages by multidisciplinary teams consisting of web designers (focused on design) and developers (focused on functionality) [25]. They allow intermixing imperative code with template content, while others such as Handlebars [16] and Mustache [13] take a simpler and cleaner approach where the templates are purely declarative. OSTRICH draws inspiration from the latter. In many such languages, it is up to the template developer to guarantee that the template will produce well-formed results. This is not a trivial task since the template itself is usually not well-formed concerning the target language grammar and thus the target language development tools cannot be used to edit and validate the template. OSTRICH addresses these concerns guaranteeing by design that only well-formed models are produced. The fact that templates are annotated model elements allows the evolution of existing tools to support defining templates.

9 CONCLUSIONS

We present an abstraction and composition mechanism for the OSTRICH template language, that targets model-driven and low-code platforms. By defining a composition mechanism for templates we allow for modular development of applications that reduces the effort of producing templates. Our developments are complementary to prior work [20, 21] in the sense that they improve the quality of the template library and the job of a template designer.

We provide a uniform composition mechanism for the OSTRICH template language that is backwards compatible with the OutSystems model and key in the model-driven composition of templates. Our instantiation algorithm significantly advances the state of the art. It accounts for a wide variety of situations with cyclic dependencies between model elements in one single pass. The semantics is based on a topological order established in a dependency graph of the different parts of model nodes. We also address the typing of template definition and template composition based on symbolic information that allows for separate phases (compile and runtime) and to produce valid runtime expressions. Finally, we evaluate our language and show that the language allows for a greater modularization of templates in an industry-standard benchmark.

Acknowledgements. Partially supported by grants UIDB/04516/2020, PTDC/CCI-INF/32081/2017, and Lisboa-01-0247-Feder-045917.

REFERENCES

[1] Atlas 2015. Atlas transformation language. https://wiki.eclipse.org/ATL/User_Guide. Last visited in 2022-05-11.

[2] Gilad Bracha. 2004. Generics in the Java Programming Language. https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf.

[3] Luís Caires and Bernardo Toninho. 2019. Refinement kinds: type-safe programming with practical type-level computation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (10 Oct. 2019). https://doi.org/10.1145/3360557

[4] Luca Cardelli. 1988. Phase Distinctions in Type Theory. (January 1988). https://www.microsoft.com/en-us/research/publication/phase-distinctions-in-type-theory/

[5] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042

[6] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. 2004. An OCL Formulation of UML2 Template Binding. In *UML 2004 — The Unified Modeling Language. Modeling Languages and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 27–40.

[7] James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.

[8] Arnaud Cuccuru, Ansgar Radermacher, Sébastien Gérard, and François Terrier. 2009. Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems* (Denver, CO) *(MODELS '09)*. Springer-Verlag, Berlin, Heidelberg, 644–649. https://doi.org/10.1007/978-3-642-04425-0_51

[9] Krzysztof Czarnecki and Simon Helsen. 2003. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Vol. 45. USA, 1–17.

[10] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. https://doi.org/10.1145/382780.382785

[11] Juan de Lara and Esther Guerra. 2013. From Types to Type Requirements: Genericity for Model-Driven Engineering. *Softw. Syst. Model.* 12, 3 (July 2013), 453–474. https://doi.org/10.1007/s10270-011-0221-0

[12] Eddy Ghabach. 2018. *Supporting Clone-and-Own in software product line*. Ph. D. Dissertation. https://tel.archives-ouvertes.fr/tel-01931217

[13] GitHub 2021. Mustache - Logic-less templates. https://mustache.github.io/. Last visited in 2022-05-11.

[14] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1 (2007), 27–34. https://doi.org/10.1016/j.scico.2007.01.013

[15] GOLEM 2020. Automated Programming to Revolutionize App Development. https://www.cmuportugal.org/large-scale-collaborative-research-projects/golem/. Last visited in 2022-05-11.

[16] Handlebars 2021. Handlebars - Minimal templating on steroids. https://handlebarsjs.com/. Last visited in 2022-05-11.

[17] JetBrains. 2020. JetBrains Meta Programming System. http://github.com/JetBrains/MPS. Last visited in 2022-05-11.

[18] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[19] Barbara Liskov and John Guttag. 1986. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, USA.

[20] Hugo Lourenço, Carla Ferreira, and João Costa Seco. 2021. OSTRICH - A Type-Safe Template Language for Low-Code Development. In *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021*. IEEE, 216–226. https://doi.org/10.1109/MODELS50736.2021.00030

[21] Hugo Lourenço, João Costa Seco, Joana Parreira, and Carla Ferreira. 2022. OSTRICH - A Rich Template Language for Low-code Development (Extended version). [under submission].

[22] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[23] OMG 2016. Meta Object Facility Specification Version 2.5.1. https://www.omg.org/spec/MOF. Last visited in 2022-05-09.

[24] OMG 2017. Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML. Last visited in 2022-05-09.

[25] Terence John Parr. 2004. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. ACM, 224–233. https://doi.org/10.1145/988672.988703

[26] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. ACM, 165–168. https://doi.org/10.1145/2500828.2500846

[27] Davide Di Ruscio, Dimitrios S. Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Softw. Syst. Model.* 21, 2 (2022), 437–446. https://doi.org/10.1007/s10270-021-00970-2

[28] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh* (proceedings of the 2002 haskell workshop, pittsburgh ed.). 1–16.

[29] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2 ed.). Addison-Wesley, Upper Saddle River, NJ. https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885/

[30] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

[31] Gilles Vanwormhoudt, Matthieu Allon, Olivier Caron, and Bernard Carré. 2020. Template based model engineering in UML. In *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*. ACM, 47–56. https://doi.org/10.1145/3365438.3410988

[32] Gilles Vanwormhoudt, Olivier Caron, and Bernard Carré. 2017. Aspectual templates in UML - Enhancing the semantics of UML templates in OCL. *Softw. Syst. Model.* 16, 2 (2017), 469–497. https://doi.org/10.1007/s10270-015-0463-3