Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01178-w

SPECIAL SECTION PAPER

Going beyond templates: composition and evolution in nested OSTRICH

João Costa Seco[1] · Hugo Lourenço[2] · Joana Parreira[1] · Carla Ferreira[1]

Received: 5 May 2023 / Revised: 24 November 2023 / Accepted: 8 April 2024

Abstract

Low-code frameworks strive to simplify and speed up application development. An essential mechanism to achieve these goals is to have native support for the safe reuse and usage of parameterized coarse-grain components, providing developers with strong guardrails and a rich software-building experience. OSTRICH—a rich template language for the OutSystems platform—was designed to simplify the use and creation of such components. Thus, the application developer can quickly reuse and assemble sophisticated and thoroughly tested application blocks. However, without a built-in composition and evolution mechanism, OSTRICH templates are still hard to create and maintain. This sometimes requires the repetition of code across different templates and creates a conflict between the customizations of the instantiated application models, and the update and reapplication of a template definition. This paper presents a principled mechanism for using abstraction in the creation of templates and simultaneously supporting the evolution of OSTRICH templates in applications after use. First, we introduce a template composition mechanism, its typing discipline, and its instantiation algorithm for model-driven low-code development environments. We start by extending OSTRICH to support nested templates and allow the instantiation (hatching) of templates in the definition of other templates. Nesting promotes a significant increase in code reuse potential, leading to a safer evolution of applications. We then introduce the support for customizable template instances, which allows one to evolve templates' code and then update a template instance without losing customizations performed in the generated code. 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, a set of annotations to support the extensions, and the corresponding validation and instantiation algorithms. In particular, we introduce a type-based validation procedure for abstractions that ensures that using templates always produces valid models. This work also extends prior developments on Nested OSTRICH with the support for safe customizations of instantiated code. We validate Nested OSTRICH using the OSTRICH benchmark by identifying the degree of reusability that can be reached in the existing sample of real templates and template uses. Our prototype is an extension of the OutSystems IDE that allows the annotation of models and their use to produce new models. We also analyze which existing OutSystems sample screen templates can be improved by using and sharing nested templates.

Keywords Metaprogramming · Low-code models · Metamodel · Templating · Typechecking templates · Model reuse

1 Introduction

Abstraction and parametrization are among the most significant mechanisms in programming languages to promote modularisation and code reuse [22]. They are present in programming languages from basic function definitions to sophisticated type systems [2] or type-level computations in metaprogramming mechanisms [3, 7].

Communicated by N. Bencomo, M. Wimmer, H. Sahraoui, and E. Syriani.

✉ Carla Ferreira
carla.ferreira@fct.unl.pt

João Costa Seco
joao.seco@fct.unl.pt

Hugo Lourenço
hugo.lourenco@outsystems.com

Joana Parreira
jb.parreira@campus.fct.unl.pt

[1] NOVA University Lisbon, NOVA LINCS, Caparica, Portugal

[2] OutSystems, Lisbon, Portugal

Published online: 20 May 2024

Low-code frameworks strive to simplify and speed up application development and abstraction is essential to achieve these goals. The native support for the safe reuse and usage of parameterized coarse-grain components in OSTRICH—a rich template language for the OutSystems platform [23, 24]. OSTRICH was designed to simplify the use and creation of such components and provide developers with strong guardrails and a rich software-building experience. The application developer can quickly reuse and assemble sophisticated and thoroughly tested application parts. However, without a built-in composition and evolution mechanism, OSTRICH templates are still hard to create and maintain. This sometimes requires the repetition of code across different templates and creates a conflict between the customizations of the instantiated application models and the update and reapplication of a template definition.

This paper is an extended version of [35], whose core contribution is to complementarily improve the template developer experience, in low-code frameworks, by allowing the reuse and composition of coarse-grain components in the definition of new templates. Previously, reapplying a template to use a newer template version or to change the arguments used, would cause all the customizations already performed on the expanded model to be lost. We further improve the experience for both template and application developers by complementing template evolution with a redo capability for code customization operations. To support composition, we equip the template language OSTRICH [23, 24] with a built-in abstraction mechanism and, to support evolution, we use a code customization logging mechanism and a redo capability. Without nested templates, OSTRICH only allowed for the top-level reuse of template application components, integrated into the IDE functionality, and any reapplication of a template would override previous instances. 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 a composition mechanism like the one introduced here, full-fledged application templates would be cumbersome to create and impractical to maintain.

OSTRICH templates follow an approach similar to other model-driven low-code programming approaches, like templates in UML (MOF) [28, 40, 41] and MetaDepth [11], where templates are instantiated in place by an external mechanism. In OSTRICH, even though the instantiation algorithm in [23, 24] is embedded into the behavior 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).

As an abstraction mechanism, OSTRICH templates represent code patterns that are commonly (re)used, allowing for greater uniformity of the code, increased safety, and thus, higher productivity. However, the most significant difference between the use of templates and other abstraction mechanisms (cf. functions) is that the result is open-box, i.e. its result is intentionally visible, and editable. According to our semantics, depicted in Sect. 8, instantiated OSTRICH templates replace the annotated anchor model elements with ground model elements (without unevaluated annotations). This means that all customizations performed to the resulting model are lost when the template is reapplied for some reason. Thus, instantiating templates or any other form of code generation is seldom the final answer to the developers' requirements. Making a generic template even more generic and equipped with a smart range of parameters does not solve the problem and usually still requires some degree of (post)customization. Customizations can be used to remove elements that are not needed but are present in the general template code, adapt properties that are not covered by the parameters, or add code that was not incorporated in the generic definition. The evolution of software produced with templates can proceed in two, until now incompatible, ways: to evolve the template definition or the arguments used to instantiate it and reevaluate its applications, or to change the generated code by hand thus losing the connection between the template and the modified instance.

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. We next extend OSTRICH to support customizable template instances by devising a bookkeeping mechanism for the customizations performed to the ground model that allows a log of operations to be reapplied, with a degree of safety, after the core instantiated model is rewritten by reapplying the template. The base mechanism that allows the reapplication of operations is the use of unique identifiers that are assigned to all OutSystems model elements. Template models use an extension of the OutSystems metamodel, thus their definitions are also tagged with unique identifiers. Instantiated model elements receive new identifiers computed from the "static" identifiers of the template node elements and the template instantiation arguments. This allows the tracking of instantiated elements and thus the reap-

plication of the customizations that were performed in the previous instantiations.

Our developments are complementary to prior work [23, 24] in the sense that they improve the experience of template developers, by improving the quality of the template library and supporting the evolution of templates, and the experience of template users by letting them edit the results without fear of losing their customizations. 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 [17] 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. Thus, the ability to change the template used is crucial. Also, machine learning techniques can 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. The same example is then used to illustrate the preservation of edits in the presence of changes in the context of the application. 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 [24], 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 (Sect. 2).
- A backward compatible representation of the model-driven composition of templates (Sect. 3).
- A one-pass instantiation algorithm that accounts for a wide variety of situations with cyclic dependencies between model elements (Sects. 4 and 5).
- An deterministic algorithm for computing identifiers of model elements in template instantiation that supports the correct reapplication of the performed customizations (Sect. 5.1).
- A typing algorithm based on symbolic information that separates phases (compile and runtime) and accounts for the composition of templates (Sect. 6).
- A declarative model for post-customizations of template instances and corresponding reapplication algorithm (Sect. 7).
- An account for modularizing templates in practice using an industry-standard benchmark (Sect. 8).

2 Nested templates

In this section, we illustrate the concept of `OSTRICH`'s nested templates by taking a new turn at modeling the application illustrated in [24] and extending and modularizing 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 [24], contain styling options in user interface components, intricate algorithms, or code patterns that developers want to get right from the start and keep uniform throughout an application. This promotes best practices on code reuse other than clone and own [14] 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 Fig. 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 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` is 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

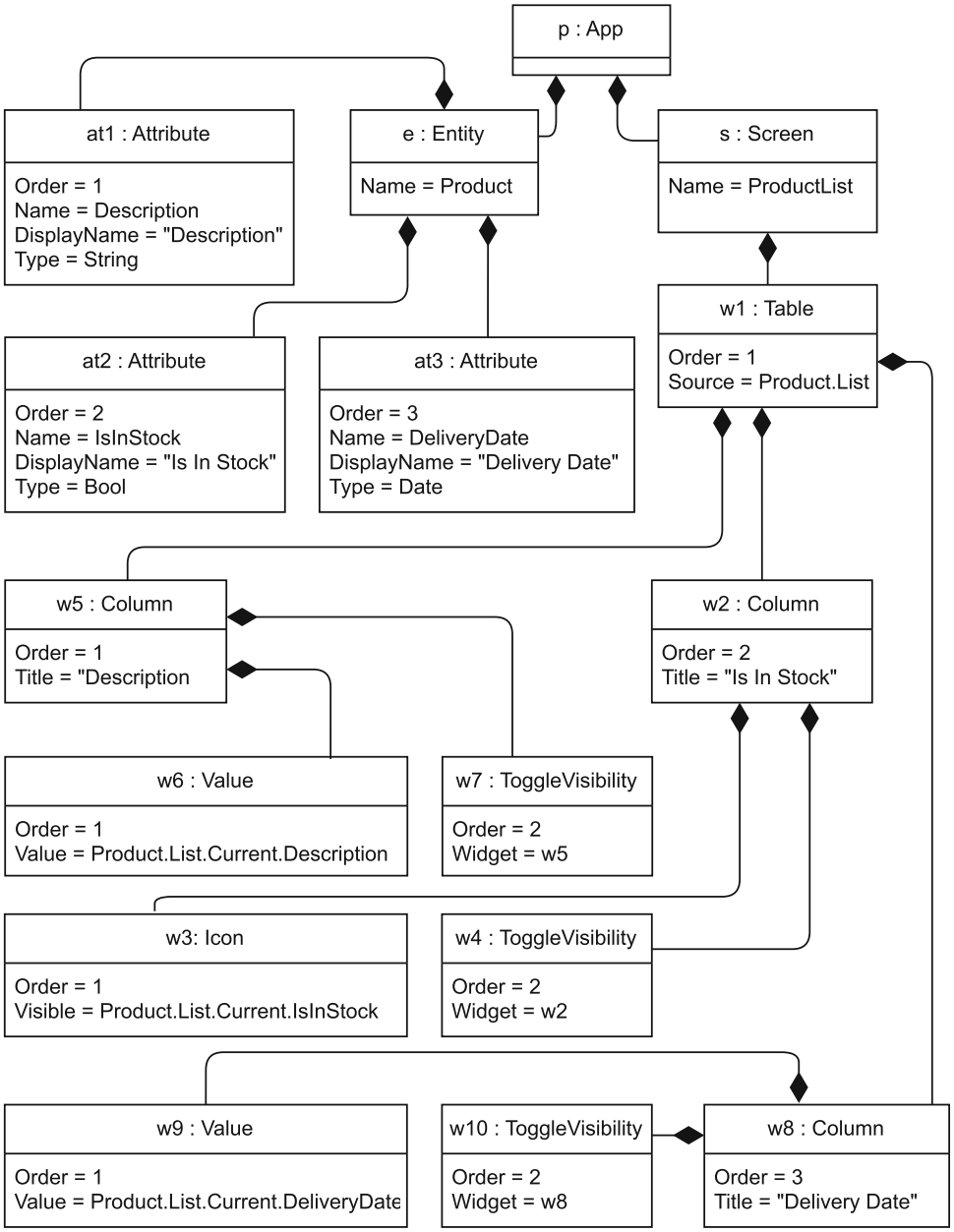


Fig. 1 The instantiated application

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 modeled in `OSTRICH` as depicted in Fig. 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-colored 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 Fig. 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 Fig. 4. In that (nested) template, `Conditional` annotations (`t3` and `t5`) condi-

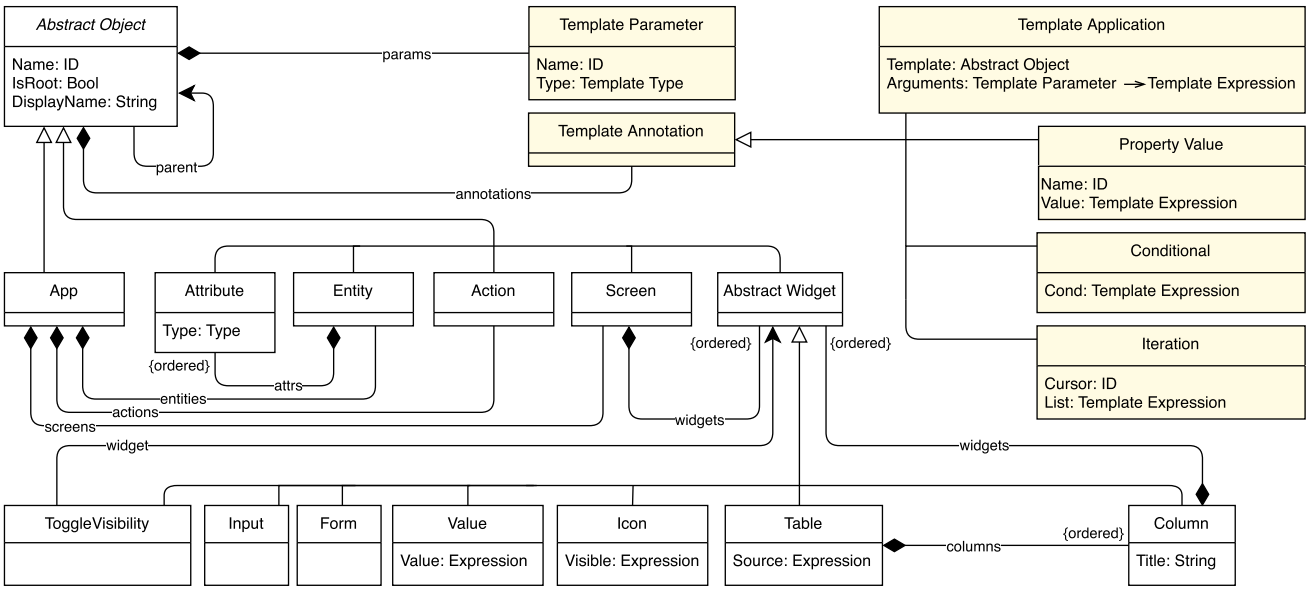


Fig. 2 The `OSTRICH` metamodel

tionally 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 [24] 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 [23] 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 Fig. 2, contains white-colored elements that correspond to the metamodel of OutSystems applications, and yellow-colored 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 Fig. 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 Fig. 4 and its children (colored 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 Fig. 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 Fig. 5, is a regular application model with an empty screen annotated with an `Template Application` annotation referring to template `p1.s1` (Fig. 3). The template preprocessor runs an instantiation algorithm and produces as a result the model in Fig. 1. The definition of nested templates allows for the reuse of the "Column" template in other scenarios, such as the one in Fig. 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).

The main challenges are related to representing standard mechanisms as the definition of abstractions and the instantiation of abstractions in a metamodel, 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

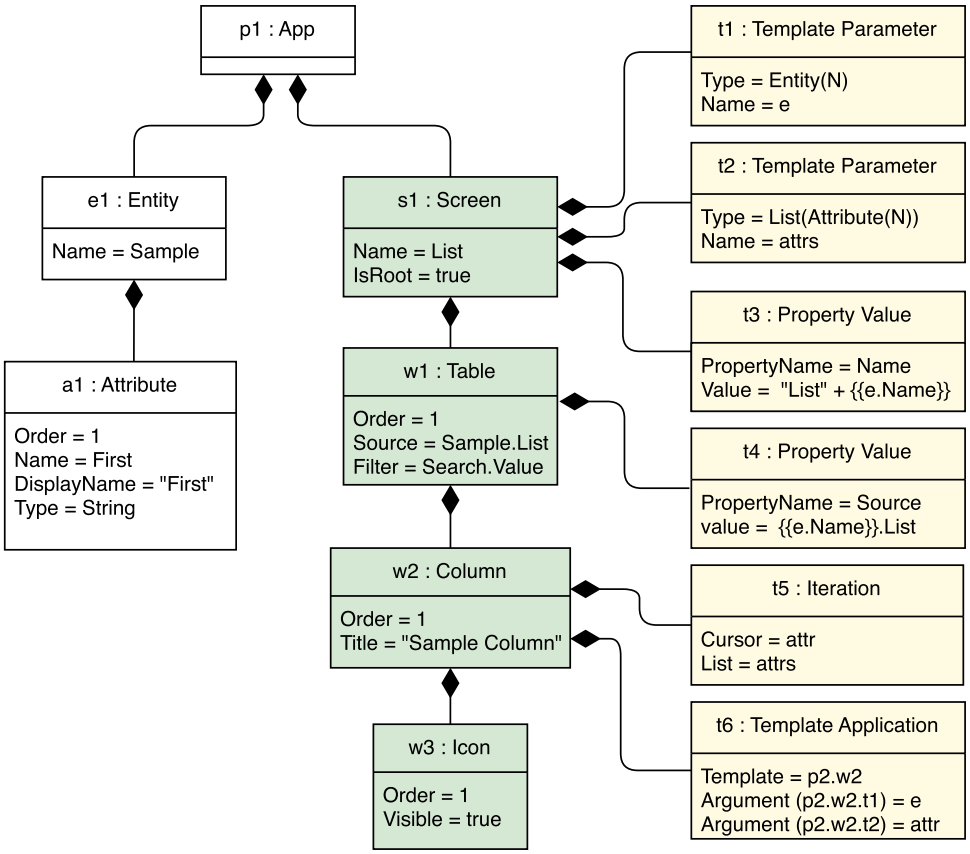


Fig. 3 A template for tables using a template for columns

using a topological order so that the result of instantiating some node can be used in another node. This is visible in Fig. 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 for assigning 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 (Fig. 7) replaces all nodes in a caller application (Fig. 8) with instances of the template nodes.

3 The template metamodel

We refine and extend the metamodel presented in [24] so that the application of a template can be uniformly used in the application models. The metamodel (Fig. 2) defines the full language of annotations that can be added to nodes in a OutSystems model. Uncolored elements correspond to (a simplified version of) the metamodel for OutSystems applications. The colored 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.

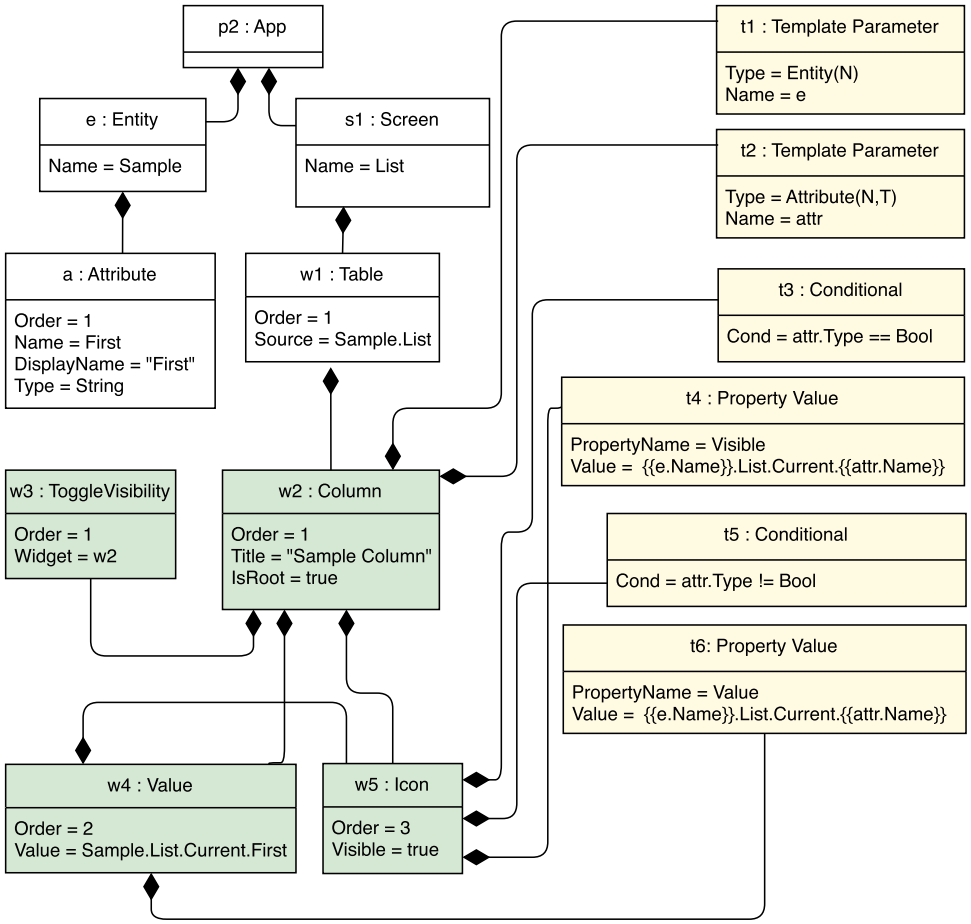


Fig. 4 The column (inner) template

- 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 [24], 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.

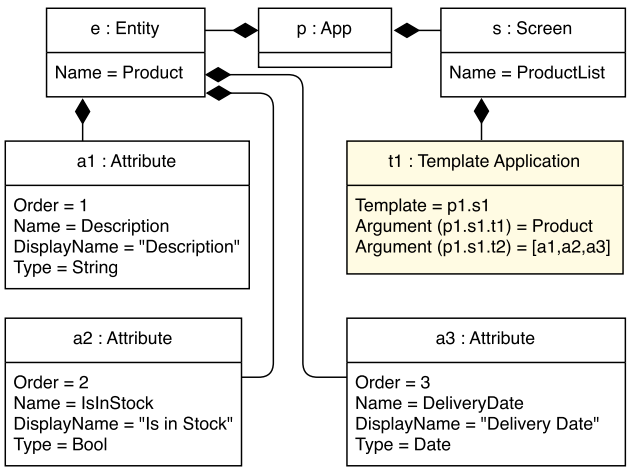


Fig. 5 The creation of the main application

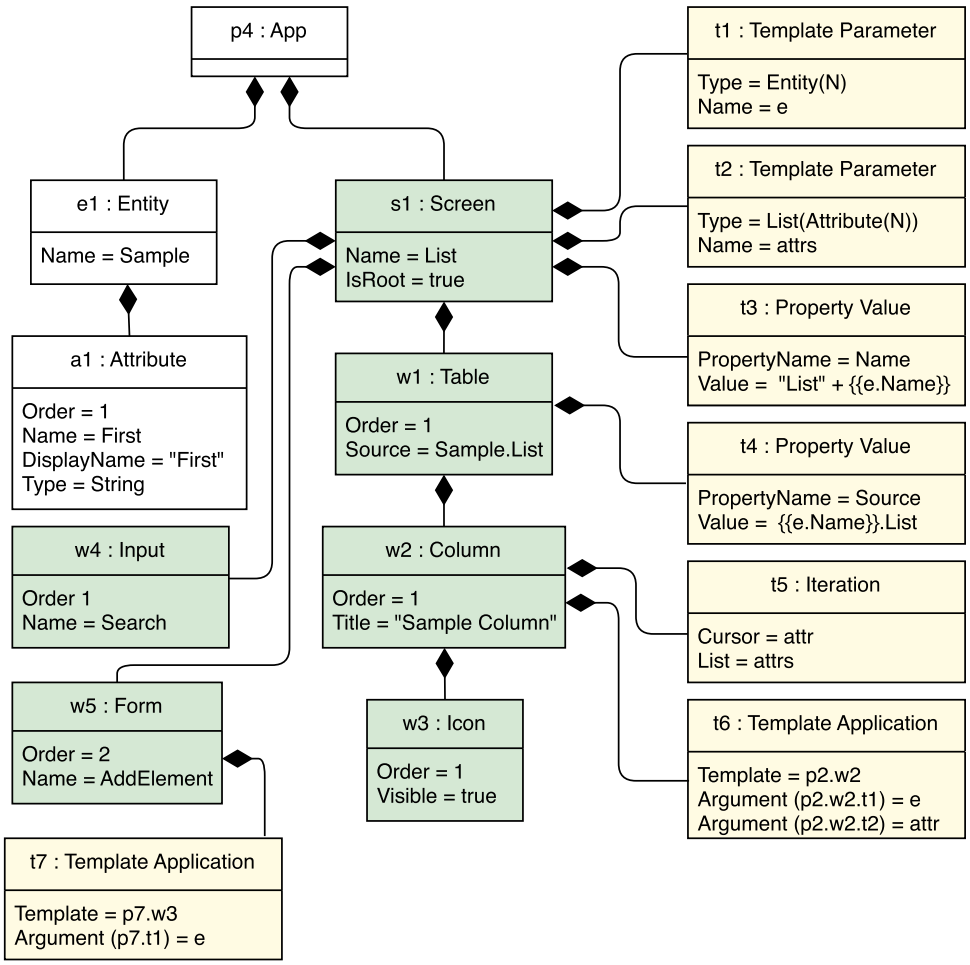


Fig. 6 Column template being reused in a more sophisticated table template, with more (sub)templates

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 Fig. 7 the name of Screen `s1` is determined by the template expression `"List"+{{e1.Name}}` in Property Value 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 Fig. 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 Fig. 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

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

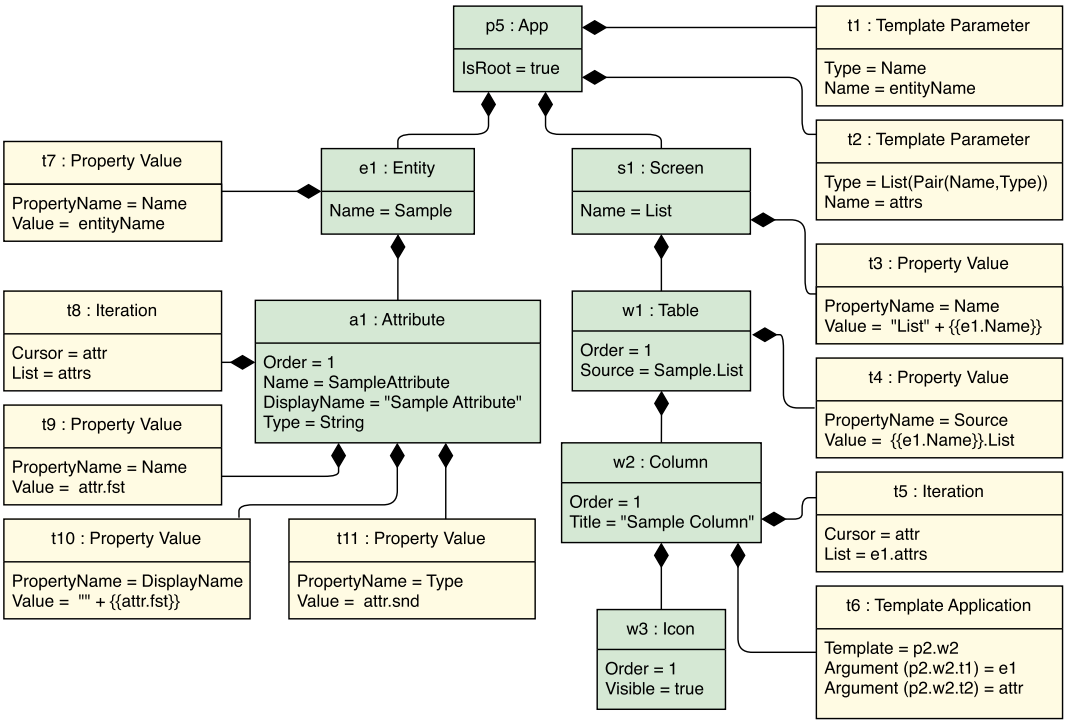
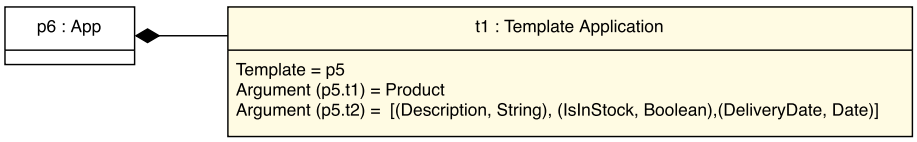


Fig. 7 A template of a complete application

Fig. 8 The creation of the main application (new)



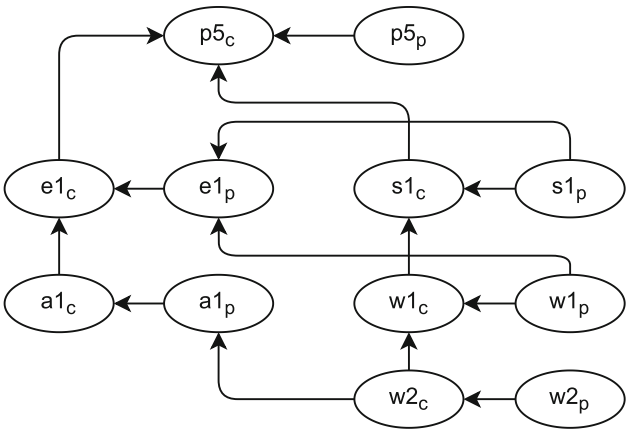


Fig. 9 Dependency graph for the template of Fig. 7

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 [24] 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 dependency can be found in Fig. 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 Fig. 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,

[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 (Part 1)

```
types
  Operation = AbstractObject × { CreateObjects, SetProperties }
  Env = { Objects: ID → Object, CursorsPosition: ID → Integer,
          Parent: × Env }
                                                          ▷ evaluation environment
  NewObjs: AbstractObject → (Env → AbstractObject)
                                                          ▷ map of template node and environment to new objects
input
  baseId: ID                                              ▷ base id for instantiated objects
  template: AbstractObject                                ▷ root template object
  targetParent: AbstractObject                            ▷ target parent object
  args: TemplateParameter → Object                        ▷ template arguments
  customizationOps: [ CustomizationOperation ]
                                                          ▷ list of customization operations
locals
  operations: Sequence of Operation                       ▷ instantiation operations
  rootEnv: Env                                            ▷ root evaluation environment
  newObjs: NewObjs
1: function INSTANTIATE(baseId, template, targetParent, args, customizationOps)
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(baseId, 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)
14:  for all op in customizationOperations do
15:    APPLYCUSTOMIZATION(op, newObjs)

input
  templNode: AbstractObject                               ▷ current template object
  targetParent: AbstractObject                            ▷ current target parent object
16: function CREATEOBJECTS(baseId, templNode, targetParent, env, newObjs)
17:   if hasConditionalAnnotation(templNode) then
18:     if evaluate(getCondExpression(templNode), env) then
19:       CREATEOBJECT(baseId, templNode, targetParent, env, newObjs)
20:   else if hasIterationAnnotation(templNode) then
21:     list ← evaluate(getListExpression(templNode), env)
22:     cursorName ← getCursor(templNode)
23:     for all (item, index) in list do
24:       newEnv ← beginScope(env)
25:       bind(newEnv, cursorName, item)
26:       bindPosition(newEnv, cursorName, index)
27:       CREATEOBJECT(baseId, templNode, targetParent, newEnv, newObjs)
28:   else CREATEOBJECT(baseId, templNode, targetParent, env, newObjs)
```

when pre-processing the model in Fig. 5, the algorithm finds the `Template Application` annotation `t1` associated to screen `s`. The corresponding table template (`p1.s1` in Fig. 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 to 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. As we will explain in detail in Sect. 7, changes made to the instantiated model are tracked so that a template can be reapplied without those changes being lost. Tracking is made possible by calculating the identifiers of instantiated objects deterministically, which in turn requires providing a base identifier for the instantiation algorithm. A simple and obvious choice is to use the identifier of the `Template Application` annotation.

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

- `baseId` = `t1` (id of the template application annotation)
- `template` = `p5` (template of Fig. 7)
- `targetParent` = `null` (annotation `t1` is applied to an app, which does not have a parent object)

Algorithm 2 Template instantiation algorithm (Part 2)

```
29: function CREATEOBJECT(baseId, templNode, targetParent, env, newObjs)
30:     id ← calculateId(baseId, getId(templNode), getCursorsState(env))
31:     if hasTemplateApplicationAnnotation(templNode) then
32:         template ← getTemplate(templNode)
33:         args ← map(getArguments(templNode),
             (param, expr) → (param, evaluate(expr, env)))
34:         INSTANTIATE(id, template, targetParent, args, [])
35:     else
36:         newObj ← createChild(id, targetParent, typeof templNode)
37:         bind(env, getId(templNode), newObj)
38:         objsInNode ← get(newObjs, templNode) ∪ { env ↦ newObj }
39:         newObjs ← newObjs ∪ { templNode ↦ objsInNode }

40: function SETPROPERTIES(templNode, newObj, env, newObjs)
41:     for all prop in getProperties(templNode) do
42:         value ← evaluateProperty(templNode, prop, env, newObjs)
43:         setPropertyValue(newObj, prop, value)

    input
      prop: Property                                          ▷ property to be evaluated
44: function EVALUATEPROPERTY(templNode, prop, env, newObjs)
45:     if hasPropertyAnnotation(templNode, prop) then
46:         return evaluate(getValueExpression(templNode, prop), env)
47:     else
48:         value ← getPropertyValue(templNode, prop)
49:         if contains(newObjs, value) then                  ▷ value is a template object
50:             objsForValue ← get(newObjs, value)
51:             if contains(objsForValue, env) then
52:                 value ← get(objsForValue, env)
53:         return value

    input
      op: CustomizationOperation                              ▷ operation to be applied
      newObjs: NewObjs                                        ▷ newly created objects
54: function APPLYCUSTOMIZATION(op, newObjs)
55:     targetObj ← lookup(newObjs, getTargetElement(op))
56:     if targetObj is null then
57:         return
58:     else if op is AddElement then
59:         newId ← getNewId(op)
60:         newElement ← getArgument(op)
61:         position ← evaluate(getPositionExpresion(op), {})
62:         cloneInto(targetObj, newId, newElement, position)
63:     else if op is RemoveElement then
64:         delete targetObj
65:     else if op is UpdateElement then
66:         prop ← getProperty(op)
67:         value ← evaluate(getValueExpression(op), {})
68:         setPropertyValue(prop, value)
```

- `args` = { `entityName` ↦ "Product," `attrs` ↦ [("Description," String), ...] }
- `customizationOperations` = `[]` (annotation `t1` does not have any customization yet)

At each point of the algorithm, there is an active *evaluation environment* used to evaluate template expressions. An evaluation environment (type `Env` in the algorithm) consists of the following:

- Objects, of type ID → Object, a map containing the objects that have been bound to the environment. These include cursor values and newly created objects (Lines 25 and 37).
- CursorsPosition, of type ID → Integer, a map containing the position of each cursor in the environment (Line 26).
- Parent, of type Env, the parent evaluation environment. As is customary, evaluation environments are organized hierarchically.

The algorithm starts by calculating the dependency graph for the template (Fig. 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 49 and 52). To bootstrap the algorithm, we initialize

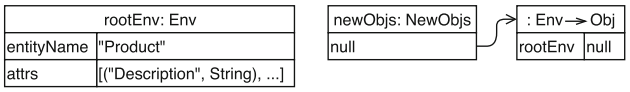


Fig. 10 Initial state of the algorithm

`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 Fig. 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 17 to 19) are processed as expected: if their condition evaluates to `true` then the template node is processed, otherwise it is ignored. For `Iteration` annotations (Lines 20 to 27) 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 28).

The `CreateObject` starts by calculating a new id by calling function `calculateId`, which we detail in Sect. 5.1. `Template Application` annotations (Lines 31 to 34) are processed by recursively calling the `Instantiate` function, using the calculated id as the base id. The creation of a new object occurs in Lines 36 to 39, using the calculated id. 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` ↦ `at1`, `env2` ↦ `at2`, `env3` ↦ `at3`}. Notice that we have three environments/objects to process, which are the result of evaluating Iteration annotation `t8` while processing operation $a1_c$. The function `SetProperties` 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` ↦ `at1` case and property `Name`. The template expression `attr.fst` (`Property Value` annotation `t9` in Fig. 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 49 to 52).

Newly created objects are stored in the evaluation environments (Line 37) so that we can properly evaluate template expressions that refer to template nodes. For instance, the template expression `e1.attrs` in `Iteration` annotation `t5` (Fig. 7) refers to the (sample) entity `e1`. When processing this expression `e1` is evaluated, as desired, to entity `e`, according to `rootEnv`. The last step of the algorithm is reapplying the customization operations (Line 14), which we present in Sect. 7.

The algorithm presented here represents a significant evolution with relation to [24] where template instantiation is external to the language, like in traditional model transformation processes [11].

5.1 Calculating identifiers

Algorithms went to the end One crucial element of the instantiation algorithm is the calculation of model element identifiers for new objects. The `calculateId` function used in Algorithm 1 is responsible for calculating identifiers for new objects in a deterministic way. This means that reapplying a `Template Application` node with the same argument values results in the same final model. Function `calculateId` starts by calculating a *compound identifier* by concatenating the following information:

- Base identifier: the identifier provided to the `Instantiate` function as argument.
- Template node identifier: the identifier of the node being instantiated.
- Cursors state: information about the active cursors in the current evaluation environment, obtained by calling function `getCursorsState`.

The function `getCursorsState` iterates the active cursors in the evaluation environment and returns the following information for each cursor:

- If the cursor value is a model object, it returns its identifier.
- If the cursor value is of type record containing a field of type ID then returns the value of that field.
- Otherwise, returns the cursor position.

The final value for the identifier is then calculated by hashing the compound identifier value. Note that for the sake of readability, we have used *simple* identifiers such as `w1` and `at1`. The actual OutSystems model, however, uses Universal Unique Identifiers (UUID) which are 128bit in size. This ensures a negligible collision probability for the hashing function.

Tables 1 and 2 illustrate the process of calculating identifiers. For example, in Table 1:

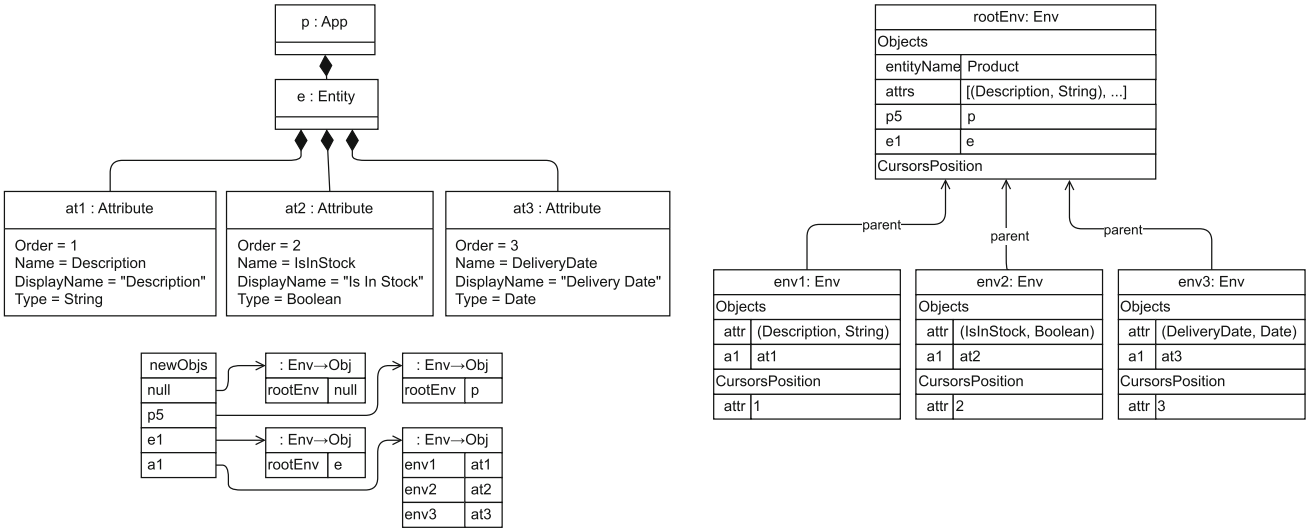


Fig. 11 Algorithm state after executing operations $p5_c$, $e1_c$, $a1_c$, and $a1_p$

Table 1 Calculating ids for template application `t1` (Fig. 8, using template of Fig. 7)

Line no	Base id	Template node id	Cursors state	Compound id	Hashed id
1	t1	p5	[]	t1:p5	p
2	t1	e1	[]	t1:e1	e
3	t1	a1	[1]	t1:a1:1	at1
4	t1	a1	[2]	t1:a1:2	at2
5	t1	a1	[3]	t1:a1:3	at3
6	t1	s1	[]	t1:s1	s
7	t1	w1	[]	t1:w1	w1
8	t1	w2	[at1]	t1:w2:at1	b1
9	t1	w2	[at2]	t1:w2:at2	b2
10	t1	w2	[at3]	t1:w2:at3	b3

Table 2 Calculating ids for template application `t6` (Fig. 7, using template of Fig. 4)

Line no	Base id	Template node id	Cursors state	Compound id	Hashed id
1	b1	w2	[]	b1:w2	w5
2	b1	w4	[]	b1:w4	w6
3	b1	w3	[]	b1:w3	w7
4	b2	w2	[]	b2:w2	w2
5	b2	w5	[]	b2:w5	w3
6	b2	w3	[]	b2:w3	w4
7	b3	w2	[]	b3:w2	w8
8	b3	w4	[]	b3:w4	w9
9	b3	w3	[]	b3:w3	w10

- Line 1 corresponds to processing operation $p5_c$. The active environment is `rootEnv`, in which there are no active cursors and thus the cursor states is the empty list. We get `t1:p5` as the compound identifier, which is hashed to `p`.[3]
- Lines 3 to 5 correspond to processing operation $a1_c$. Template node a1 contains an `Iteration` annotation (t8,

[3] Remember that we are using simpler identifiers for the sake of simplicity. The actually hashed identifiers are much longer.

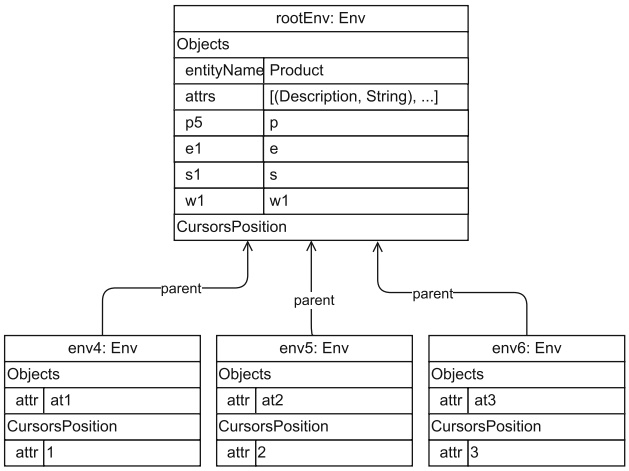


Fig. 12 Evaluation environments created after executing operation $w2_c$

Fig. 7), which results in three evaluation environments being created: env1, env2, and env3. The only active cursor in these environments is `attr`. Its value is neither a model object nor a record containing a field with label ID, and thus we use its position as the cursor state.

- The last three lines correspond to processing operation $w2_c$. Template node w2 also contains an `Iteration` annotation (t5, Fig. 7), which results in three additional evaluation environments of Fig. 12. The only active cursor in these environments is `attr`. Its value is each of the entity attributes created by $a1_c$, respectively at1, at2, and at3. Consequently, we use those objects' identifiers as the cursor state. Note that, in this case, the obtained identifier is not directly the identifier of an object: template node w2 also contains a `Template Application` annotation (t6, Fig. 7. The obtained identifiers are instead used as the base identifiers for each recursive call to the `Instantiate` function, as illustrated by Table 2.

The approach that we follow to calculate the compound identifiers ensures that no repeated identifiers are obtained, which is required by the OutSystems model. We represent the cursor state by an identifier field whenever possible so that when reapplying a template we can preserve as much as possible of the existing model. Note that by default the cursor position is used as its state, which in cases such as our example of Fig. 8 means that reapplying the template with a new list of arguments may result in the wrong attribution of identifiers. Our algorithm accounts for an easy solution, though. By changing the type of template parameter `t2` of Fig. 7 to `List(Triple(ID, Name, Type))`, an explicit identifier value can be provided when instantiating the template. In visual IDEs such as the OutSystems IDE such identifiers can be automatically managed by the IDE, freeing the developer from maintaining them.

6 Checking model soundness

In addition to the instantiation algorithm embedded in the pre-processing phase of the publication process in ODC Studio [30], we define a verification that checks the preconditions 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 Sect. 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 Fig. 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, type-checking 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 Fig. 13, 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 Fig. 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 (Fig. 7), it is represented by `Entity(Product)`, and its attribute by type

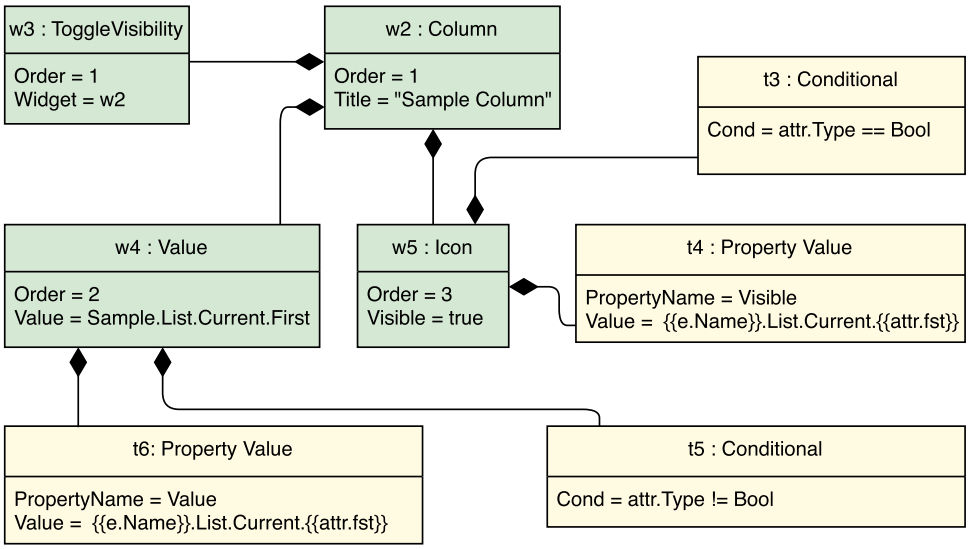


Fig. 13 A node with alternative child node types

`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` is the type of the values it represents. We then extend this language of types with standard types. Records of elements have type $\{L_i : T_i^{\,i \in 1 \ldots n}\}$, 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 [23] 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 Sect. 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.[4] We present a series of examples as accessory materials in the repository.

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 [26]. Take the example of the template application in Fig. 3 that instantiates the template `p2.w2` in Fig. 4. The inner template (Figure 4) is parametrized by the compile-time values with the entity type `Entity(N)` and the attribute type `Attribute(N, T)`. In Fig. 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 Fig. 3. This approach is sound due to the immutability of the attribute lists. Additionally, note that, in Figs. 3 and 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 $\texttt{N} \mapsto \texttt{N'}$ and $\texttt{T} \mapsto \texttt{Top}$, which is a valid substitution and ensures the correctness of the template application in `p1.w2.t6` (Fig. 3).

The (formal) functional implementation of the algorithm uses universal quantification to declare such names and types rather than unification. In a low-code setting, it is not expected developers to explicitly specify type variables. In

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

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 Fig. 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 3 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 Fig. 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, the syntax with double curly braces [24] identifies compile-time expressions embedded into runtime expressions. Notice that `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 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 3 shows a fragment of the type-checking 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 type-checking 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 Customizable templates

We now present the customization mechanism that allows the tracking and reapplication of editing operations performed after the instantiation of a template. In an editing environment where the edition of instantiated templates is seamless, i.e. equivalent to editing the generated code as built by hand, it is important to allow the reapplication of the templates without losing the customizations performed by the developer. Scenarios of the reapplication of a template include the changing of the arguments used, for instance, data sources or lists of attributes to be shown. Template reapplication is also particularly important when combining the abstraction provided by templates with other, more automated, programming mechanisms being developed by OutSystems that allow one to alternate between known alternatives for a particular widget in the application [37].

Algorithm 3 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
```

Take the example depicted in Fig. 1, which is the target model obtained from instantiating the template from Fig. 7 with the model present in Fig. 8. One simple and natural customization is to change the title of column `w8` from "Delivery Date" to "Expected Delivery Date". This can be done in two different ways. It is possible to change the annotation node `t1` in Fig. 8, and modify the value in the list of attributes to be created. But, in the context of an interactive and visual system, the direct editing of the title property is not only possible but also simpler and more convenient for the developer. The subsequent edit is to add an icon to the column of the delivery date with an expression that checks if the delivery date is less than a week from the current date and shows an alert sign. This is only possible by changing the template with more code to cover this case, hence impacting all future uses of the template, which may not be intended.

Figure 14 shows the extension of the `OSTRICH` metamodel (Fig. 2) to support the tracking of customizations. Associated with a template application annotation, we have a sequence of operations of three possible types: adding an element, removing an element, and updating a property value of an element. All operations store the identifier of its target element, declared in the superclass `Customization Operation`. Then, the `Add Element` operation provides its main argument, the element to be added, the position in which it has been added to the list of children of the parent node, and the identifier that was assigned to it. All the information stored is crucial so that an `Add Element` customization can be precisely replicated. The `Remove Element` operation does not need any parameter other than the target element. The `Update Element` operation specifies the property and the value to be updated.

In the case of our running example, the modified model, with the customizations depicted in red and the corresponding log of operations in blue, is shown in Fig. 15. Notice that the title of node `w8` changed, and a new node was added (`w11`). We extend our semantics with relation to [35] so that the application node (`p`) keeps the `Template Application` annotation after its instantiation, and also keeps a record of the sequence of operations that were performed in the IDE under the instance's root node. In this case, the operations are represented by model elements `o1` and `o2`. Notice that operation `o2` has an extra element connected to it, which is a copy of the element added to the active model. In this case, the identifier of the target element (`w8`) is retrieved from the editing operation in the IDE by pointing and clicking on the element. In practice, the set of operations depicts the differences between the model that was obtained by code generation and the current model and need not be the literal operations of the IDE.

Consider now that a new attribute called "Delivery Notes" is added to the arguments of the application annotation node (`t1`), depicted in Fig. 16. A related effect would be achieved if the parameters were a database entity and their fields had changed, as in the case of the template in Fig. 6. This edit invalidates the instantiated model under the application (`App`) node `p`.

To bring the model up to date, the template should be reinstantiated and thus be updated with the new set of model elements and the customizations should be applied to merge the two updates to the model (changing parameters and editing). In Algorithm 1, this is accomplished at Line 14, which calls the `ApplyCustomization` function (Line 54) for each of the customization operations. The function starts by looking up the target element for the customization operation,

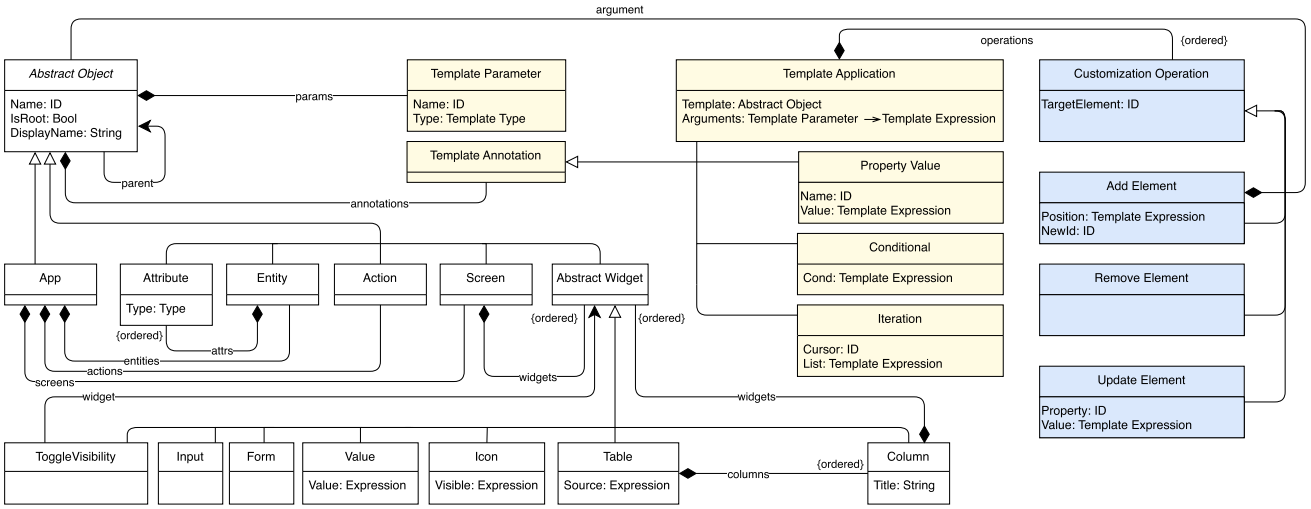


Fig. 14 Metamodel depicting customization operations

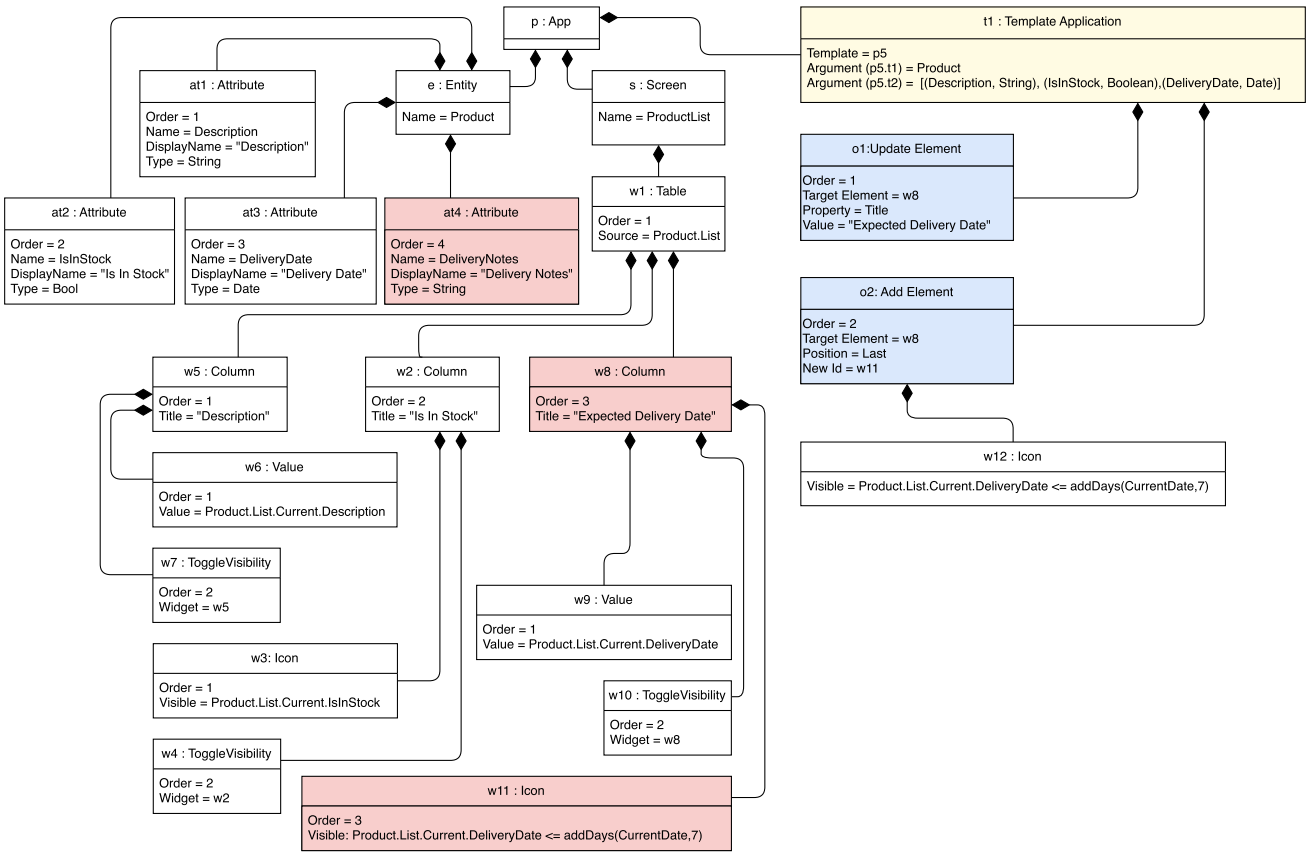


Fig. 15 Target model with customization operations

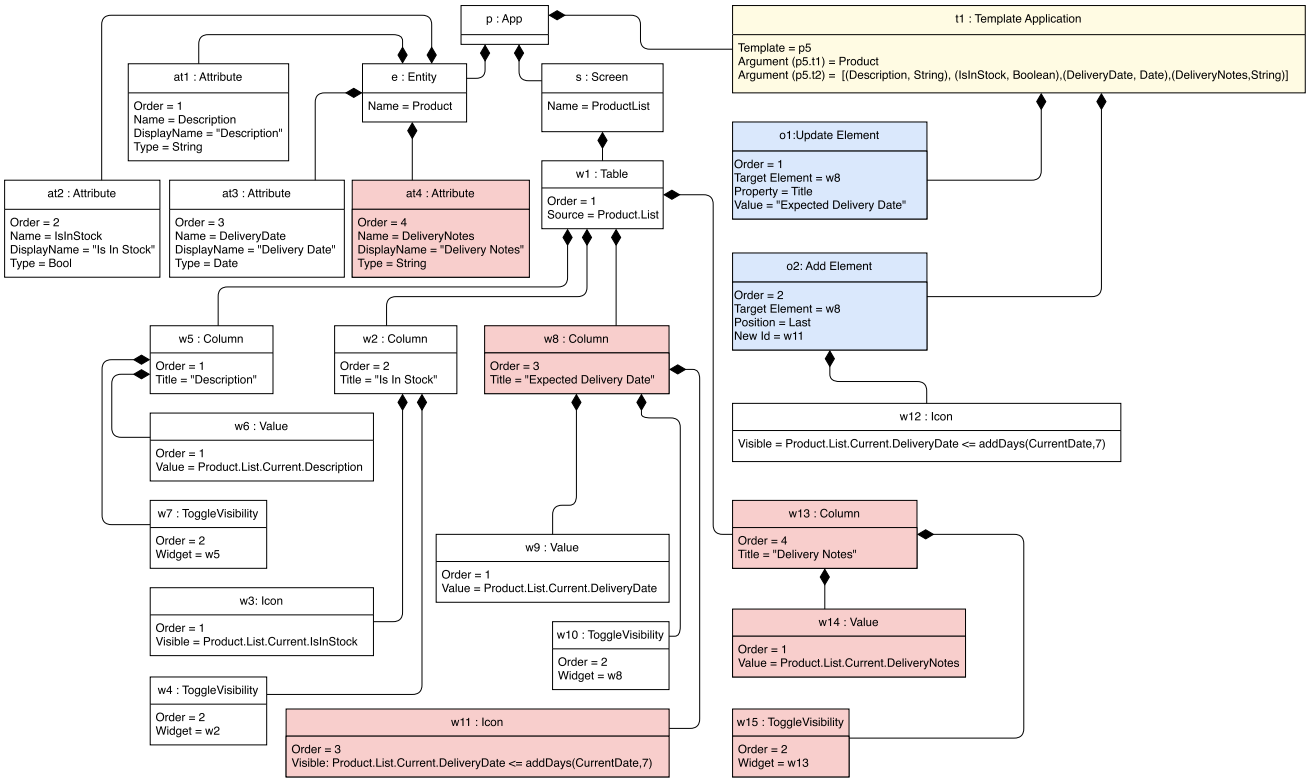


Fig. 16 Target model after reapplication of the template

and immediately returns if the object does not exist. Otherwise, it changes the model according to the customization operation properties.

As presented in Sect. 5.1, our algorithm uses a deterministic strategy to calculate model element identifiers. Since the additional attribute was added to the end of the Template Application annotation arguments list, all the model elements that existed in the initial instantiation of the template will still exist in the reinstantiated template with the same identifiers. So, all customizations will still refer to the correct model elements and can be reapplied without any problems. Figure 16, the add and update operations are straightforwardly applied to the element with identifier `w8` and the resulting model preserves the edits performed by the developer. It contains both the modified title, the added widget, and the new column.

In this case, an operation refers to an element that is no longer produced by the template instantiation, the operation is ignored not reapplied. For instance, consider that instead of adding the new attribute `DeliveryNotes` we had instead removed attribute `DeliveryDate`, i.e., we modified the arguments of node `t1` of Fig. 16 to consist only of attributes `Description` and `IsInStock`. In this case, after reapplying the template some elements that had been generated in the initial template application would no longer be generated, namely entity attribute `at3` and column widget `w8`. In this scenario, operations `o1` and `o2` would no longer be applicable and would be ignored. They would, however, still be kept in the operations list—it is always possible for the developer to go back and change the template application arguments to reintroduce the removed attributes, in which case the operations become applicable again.

The use of identifiers captured from the user interaction is a simple and direct way of storing and localizing the operations in the model. The base mechanism that allows the reapplication of operations is the use of unique identifiers that are statically assigned to all OutSystems model elements. This is only possible because the instantiation algorithm in `OSTRICH` generates new identifiers for the new elements in a deterministic way (Sect. 5.1), which means that all operations will be applied in the correct context.

This also means that the template developer should take care to guarantee that reapplying the template will not result in identifier mix-ups. In our example, nodes `at1`, `at2`, `at3`, `at4` correspond to the (generated) entity attributes `Description`, `IsInStock`, `DeliveryDate`, and DeliveryNotes, respectively. These identifiers have been generated in part from scalar values: the cursor index (see Table 1). In turn, these identifiers contributed to the calculation of identifiers `w5`, `w2`, `w8`, and `w13` (see Table 1), which correspond to the table widget columns. Notice that operations `o1` and `o2` in Fig. 15 refer to column `w8`. If we had included the new attribute `DeliveryNotes` in any position other than

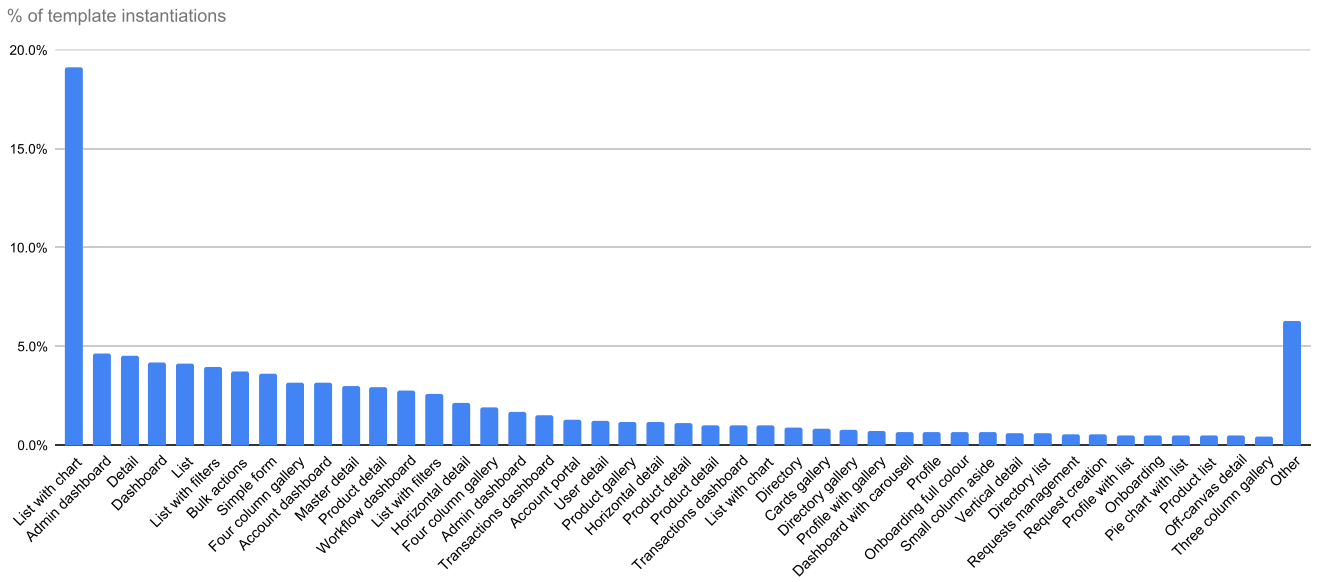


Fig. 17 Template instantiations

the last one in the Template Application arguments (node `t1`, Fig. 16) we would still get the same identifiers but they would refer to conceptually different attributes in the entity and columns in the table widget, and thus operations `o1` and `o2` would be applied to the wrong column. As mentioned in Sect. 5.1, the template developer can prevent the cursor index from contributing to the identifier calculation by requiring an explicit identifier to be provided (cf. react-like keys in new elements). Such explicit identifiers can be automatically provided by the IDE and thus do not require extra effort for the developers using the template.

Also, the position of the new elements in the list of children of a node is not always deterministic and can have undefined behaviors if the new template does not produce the same reference element as in previous instantiations. In some situations, we can use default fallback values, like the last position in the list of elements, or just ignore the effect of the operation entirely. For instance, eliminating an element that is no longer present has the same effect if ignored, and adding a new child element to an element that no longer exists in the instance, does not make sense and can be ignored. A warning may be issued for user awareness.

8 Evaluation

We evaluate the new version of `OSTRICH` by looking for shared patterns in existing OutSystems screen templates. We started by taking a random sample of 120K template instantiations over a span of four years of generalized use in the platform (Fig. 17), and then looked at the top ten (out of 70) most used templates (Table 3). Collectively, this set of ten templates accounts for more than 50% of the screen template instantiations that we sampled. A long tail dropping significantly after the first ten can be observed in the usage profile of templates depicted in Fig. 17.

A total of seven shared patterns were discovered (Table 4) and represented as nested `OSTRICH` templates. The existing screen templates were successfully modified to take advantage of the new nested templates. We depict, in Fig. 18, the reuse of templates in the new template definitions. The initially existing screen templates are on the left-hand side of Fig. 18, and the new nested templates are depicted and highlighted on the right-hand side of the figure. The relations between templates depict the usage of a nested template in the definition of the other.

Finding seven shared patterns in this small set of screen templates is quite significant, and demonstrates the level of reuse that can occur 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 Fig. 4, which is a simplified variant of the `Attribute` template and in which only the Boolean type is handled. OutSystems has a total of 12 different basic types with distinct visualization rules that usually require specialized code, not needed when using an `Attribute` template. The addition of seven shared patterns to the initial set represents an increase in our library of templates from ten to 17 (a 70% increase).

We have analyzed a small but significant subset of screen templates (the top ten out of 70), but we anticipate that many

Table 3 Top ten most used "screen templates"

Screen template	% of total instantiations
List with charts	19.1
Admin dashboard	4.7
Detail	4.6
Dashboard	4.2
List	4.2
List with filters	4.0
Bulk actions	3.7
Simple form	3.7
Four column gallery	3.2
Account dashboard	3.2
Total	54.6

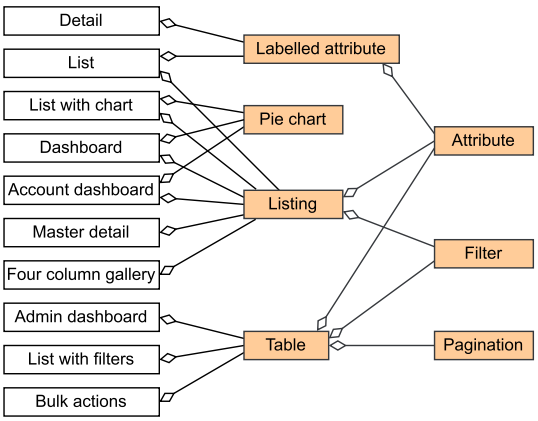


Fig. 18 Screen templates using the new nested templates

more shared patterns will be found in the remaining screen templates. User-defined patterns and templates are at the core of new developments in low-code platforms such as OutSystems [25, 31], where a power user can define new patterns and templates to be used by other users and also virtualization and AI-powered synthesis mechanisms.

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 Fig. 7.

The existing (nonparameterized) templates that we evaluated are all screen templates, i.e., they are used to create screens and their user interface (UI) elements. As our running example shows, OSTRICH is not limited to creating user interface elements. However, a considerable fraction of an OutSystems module consists of UI elements (Table 5). We sampled 897 OutSystems modules, of which 39.9% contained at least one screen,[5] with each screen containing an average of 228.5 UI elements. The histogram of Fig. 19 depicts the distribution of the quantity of screens per number of UI elements for our sample of 3681 screens. Screens and their UI elements correspond to 41% of the model elements in the sampled modules (59% if we consider only the modules with at least one screen), which established an upper bound on the developer effort that can be removed if every screen is created from a template. We currently do not keep information about which template (if any) was used to create a screen and thus are unable to provide a lower bound. However, with the introduction of OSTRICH nested templates and the ability to customize their instances we will have, in the future, more information that allows us to properly assess the impact of OSTRICH on the development time.

9 Related work

Model driven engineering As stated in [12], 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 [38] or MPS [20, 33] define transformations as external to the language itself (cf. ATL [1]). Moreover, the soundness guarantees provided the type-checking 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 [39]. To support the algorithmic construction of programs at compile time, multistage programming has been for several mainstream functional programming languages, notably MetaML [39], MetaOCaml [21], and Template Haskell [36]. 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

[5] OutSystems apps are usually split into modules dedicated to UI, logic, and data, which explains why modules with screens are a minority.

Table 4 Nested templates

Nested template	Description
Labeled 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 5 OutSystems apps complexity

Number of modules	897	
Number of modules with screens	358	39.9%
Number of screens	3681	
Number of UI elements	841,218	
Average number of UI elements per screen	228.5	
Number of model elements in all modules	2,051,985	
Number of UI elements in all modules	841,218	41.0 %
Number of model elements in modules with at least one screen	1,425,754	
Number of UI elements in modules with at least one screen	841,218	59.0%

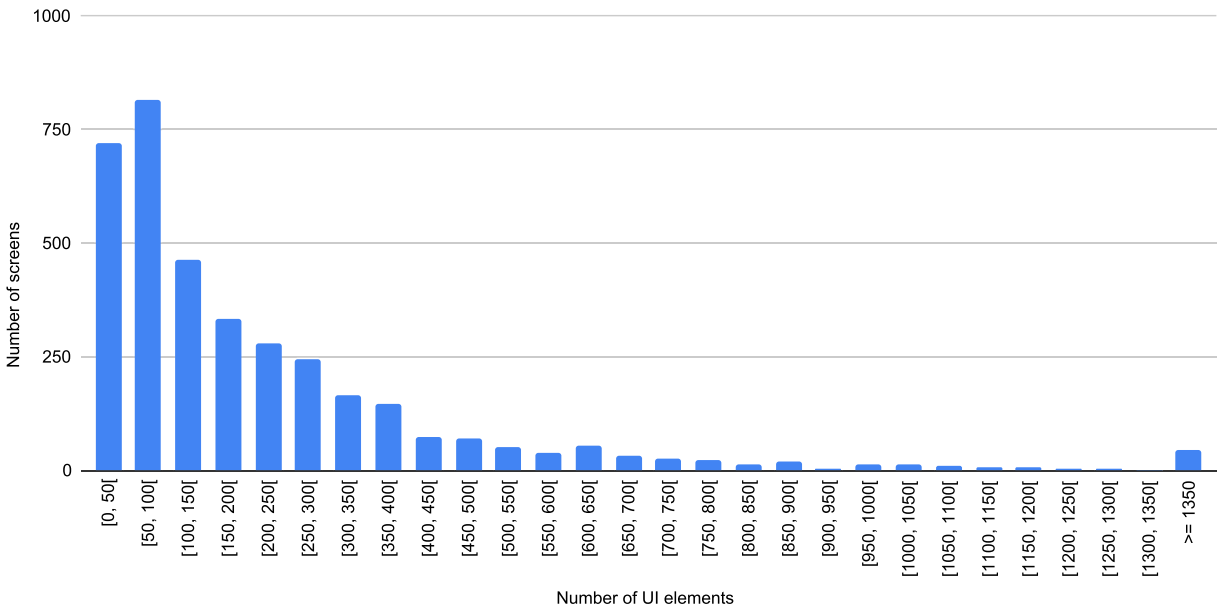


Fig. 19 Screen complexity

type-level computations, in a low-code context. Crucially, we developed a typing algorithm to analyze 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 [16, 40], model conformance is performed on the instances after parameter substitution. `OSTRICH` checks conformance statically by verifying the template and its arguments at compile-time.

Program synthesis and live programming

Typing and synthesis approaches based on holes in programs [18, 27] pursue the same goal of replacing a subtree of the original AST with compatible expressions. This is similar to our approach of replacing a node with its re-instantiation of a template. A special representation is needed for the model to act as a proper low-level model and a model that contains extra information to be handled in the dynamic changes of the code. Complementary to these approaches, works with concerns directed to reconfiguration and the constant evolution of code [13, 34] take similar approaches to ours in the sense that they also use a queue of operations to be (re)applied to the volatile core code. We take a more flexible approach by allowing for some operations to not have an effect if the supporting elements are dynamically missing.

The combination of these two approaches with the reapplication of customizations is useful in the setting of low-code automated platforms like [37] where different template alternatives can be synthesized from evolving inputs.

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 behavior customization in the instantiated code.

UML templates Templates in UML [29] address model reuse through the concepts of abstraction and parametrisation [22], with some variants proposed and instantiated in EMF-based tools [6, 40, 41]. 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 [16], like in [41], or using contracts [8]. However, in both approaches [8, 41], 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) [32]. They allow intermixing imperative code with template content, while others such as Handlebars [19] and Mustache [15] 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.

10 Future work

Our approach allows a template to be reapplied even after the result of the initial instantiation has been customized by the developer. We achieve this by tracking the changes made to the instantiated model in a list of operations that are optimistically reapplied whenever the template is reinstantiated. Initial results are promising, but need to be subject to larger-scale usability testing to confirm the usefulness of the approach.

We have considered an alternative strategy that builds upon the existing merge mechanisms for OutSystems models, namely three-way merge. Recall that a three-way merge is used when we are given two models evolved from a common base model. That is, given a base model `Source` and two models `Mine` and `Other` resulting from editing model `Source`, a three-way merge algorithm can integrate changes made in the model `Other` into the model `Mine` in a mostly automated fashion. In our scenario, if we take model `Source` to be the model obtained by the initial template instantiation, model `Mine` the one resulting from the developer customization to model `Source`, and model `Other` the one resulting from reapplying the template with new argument values, by applying the three-way algorithm to these three models we can in theory preserve the developer changes without having to keep an explicit list of their operations. We plan to implement this alternative and compare it to the one presented in this paper.

OutSystems product roadmap heavily focuses on AI-assisted development [31]. `OSTRICH` will be instrumental in several different steps of the developer journey. For instance, when creating a new application from a human-provided prompt the AI engine can work at a higher abstraction level by selecting from a set of existing parameterized templates and choosing their argument values, rather than having to produce a complete description of the application's code.

11 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 [23, 24] 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 backward 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.

Funding Open access funding provided by FCT|FCCN (b-on).

References

1. Atlas: Atlas transformation language. https://wiki.eclipse.org/ATL/User_Guide. Last visited in 2022-05-11 (2015)
2. Bracha, G.: Generics in the Java programming language. https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf (2004)
3. Caires, L., Toninho, B.: Refinement kinds: type-safe programming with practical type-level computation. Proc. ACM Program. Lang. **3**(OOPSLA), 1–30 (2019). https://doi.org/10.1145/3360557
4. Cardelli, L.: Phase distinctions in type theory. https://www.microsoft.com/en-us/research/publication/phase-distinctions-in-type-theory/ (1988)
5. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–523 (1985). https://doi.org/10.1145/6041.6042
6. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G.: An OCL formulation of UML2 template binding. In: UML 2004—The Unified Modeling Language. Modeling Languages and Applications. Springer, Berlin, pp. 27–40 (2004)
7. Cheney, J., Hinze, R.: First-class phantom types. Technical Report. Cornell University (2003)
8. Cuccuru, A., Radermacher, A., Gérard, S., Terrier, F.: 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), pp. 644–649. Springer-Verlag, Berlin (2009). https://doi.org/10.1007/978-3-642-04425-0_51
9. Czarnecki, K., Helsen, S.: 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, pp. 1–17. USA (2003)
10. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001). https://doi.org/10.1145/382780.382785
11. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Model. **12**(3), 453–474 (2013). https://doi.org/10.1007/s10270-011-0221-0
12. Di Ruscio, D., Kolovos, D.S., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: two sides of the same coin? Softw. Syst. Model. **21**(2), 437–446 (2022). https://doi.org/10.1007/s10270-021-00970-2
13. Domingues, M., Seco, J.C.: Type safe evolution of live systems. In: Workshop on reactive and event-based languages & systems (REBLS'15) (2015). https://docentes.fct.unl.pt/jrcs/files/rebls15.pdf
14. Ghabach, E.: Supporting clone-and-own in software product line. Ph.D. Dissertation (2018) https://tel.archives-ouvertes.fr/tel-01931217
15. GitHub: Mustache—Logic-less templates. https://mustache.github.io/. Last visited in 2022-05-11 (2021)
16. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1), 27–34 (2007). https://doi.org/10.1016/j.scico.2007.01.013
17. GOLEM: Automated Programming to Revolutionize App Development. https://www.cmuportugal.org/large-scale-collaborative-research-projects/golem/. Last visited in 2022-05-11 (2020)
18. Guerra, H., Ferreira, J.F., Seco, J.C.: Hoogle⋆: synthesis of constants and λ-abstractions in Petri-net-based Synthesis using symbolic execution. In: 37th European Conference on Object-Oriented Programming, ECOOP 2023 (2023)
19. Handlebars: Handlebars—minimal templating on steroids. https://handlebarsjs.com/. Last visited in 2022-05-11 (2021)
20. JetBrains: JetBrains Meta Programming System. http://github.com/JetBrains/MPS. Last visited in 2022-05-11 (2020)
21. Kiselyov, O.: 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), pp. 86–102. Springer (2014) https://doi.org/10.1007/978-3-319-07151-0_6
22. Liskov, B., Guttag, J.: Abstraction and Specification in Program Development, p. 0262121123. MIT Press, Cambridge (1986)
23. Lourenço, H., Seco, J.C., Parreira, J., Ferreira, C.: OSTRICH: a rich template language for low-code development (extended version). Softw. Syst. Model. (2022). https://doi.org/10.1007/s10270-022-01066-1

24. Lourenço, H., Ferreira, C., Seco, J.C.: 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, pp. 216–226. IEEE (2021). https://doi.org/10.1109/MODELS50736.2021.00030
25. Lourenço, H., Seco, J.C., Ferreira, C., Simões, T., Silva, V., Assunção, F., Menezes, A.: CHAMELEON: OutSystems Live Bidirectional Transformations (2023). arXiv:2305.03361
26. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
27. Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. Proc. ACM Program. Lang. **3**(POPL), Article, 14, 32 (2019). https://doi.org/10.1145/3290327
28. OMG: Meta Object Facility Specification Version 2.5.1. (2016). https://www.omg.org/spec/MOF. Last visited in 2022-05-09
29. OMG: Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML. Last visited in 2022-05-09 (2017)
30. OutSystems: OutSystems Developer Cloud (2023). https://success.outsystems.com/documentation/outsystems_developer_cloud/getting_started_with_odc/ui_overview_of_odc_portal_and_odc_studio/
31. OutSystems: Project Morpheus—Generative AI (2023) https://www.outsystems.com/news/generative-ai-roadmap/
32. Parr, T.J.: 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, pp. 224–233. ACM (2004) https://doi.org/10.1145/988672.988703
33. Pech, V., Shatalin, A., Voelter, M.: 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, pp. 165–168. ACM (2013). https://doi.org/10.1145/2500828.2500846
34. Seco, J.C., Caires, L.: Types for dynamic reconfiguration. In: Sestoft, P. (ed.), Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings (Lecture Notes in Computer Science, vol. 3924), pp. 214–229. Springer. (2006). https://doi.org/10.1007/11693024_15
35. Seco, J.C., Lourenço, H., Parreira, J., Ferreira, C.: Nested OSTRICH: hatching compositions of low-code templates. In: Syriani, E., Sahraoui, H.A., Bencomo, N., Wimmer, M. (eds.), Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23–28, pp. 210–220. ACM (2022). https://doi.org/10.1145/3550355.3552442
36. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 Haskell Workshop, Pittsburgh, pp. 1–16 (2002)
37. Seco, J.C., Aldrich, J., Carvalho, L., Toninho, B., Ferreira, C.: Derivations with holes for concept-based program synthesis. In: *Proceedings of the 2022 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software* (Auckland, New Zealand) (Onward! 2022), pp. 63–79. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3563835.3567658
38. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2 edn.). Addison-Wesley, Upper Saddle River (2009). https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885/
39. Taha, W., Sheard, T.: 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), pp. 203–217. Association for Computing Machinery, New York (1997). https://doi.org/10.1145/258993.259019
40. Vanwormhoudt, G., Allon, M., Caron, O., Carré, B.: 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, pp. 47–56. ACM (2020). https://doi.org/10.1145/3365438.3410988
41. Vanwormhoudt, G., Caron, O., Carré, B.: Aspectual templates in UML—enhancing the semantics of UML templates in OCL. Softw. Syst. Model. **16**(2), 469–497 (2017). https://doi.org/10.1007/s10270-015-0463-3

Carla Ferreira received a PhD from the University of Southampton (2003). She is an Associate Professor at NOVA University Lisbon and a Researcher at the NOVA LINCS Research Centre in Portugal. She currently coordinates TaRDIS —a Horizon Europe project on programming tools for decentralized swarms. Her research is concerned with developing formal calculi, techniques, and tools to express and reason about concurrent and distributed systems, with the overall goal of helping programmers build trustworthy systems. She has published in top-tier venues, including POPL, VLDB, EuroSys, OOPSLA, ESOP, MODELS, and CONCUR.

Hugo Lourenço graduated in Computer Engineering at Instituto Superior Técnico, Portugal. He has been at software engineering positions for most of his career. Currently, he is a Distinguished Research Engineer at OutSystems. His main responsibilities center around the definition and evolution of the OutSystems visual language and its metamodel.

João Costa Seco graduated in 1993, got a Masters in 1997 and got his PhD from NOVA in 2006. He is a Researcher at the Software Systems group of NOVA LINCS and an Assistant Professor at NOVA Science and Technology School, NOVA University Lisbon. His research, teaching, and knowledge transfer activities are centered on the use of programming language-based approaches for automated programming and software evolution to better enable software development processes, advance the state of the art, and improve software engineering practices. He actively participates in applied research projects and collaborative research initiatives with the industry.

Joana Parreira got an MSc degree from the NOVA School of Science and Technology, NOVA University Lisbon (2022). During her master's, she formalized and implemented a core template language for OSTRICH, a type-safe template language for OutSystems. She is currently working as a Software Engineer at Microsoft.