Software and Systems Modeling (2023) 22:1645–1663
https://doi.org/10.1007/s10270-022-01066-1

SPECIAL SECTION PAPER

OSTRICH: a rich template language for low-code development (extended version)

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

Received: 16 March 2022 / Revised: 18 September 2022 / Accepted: 3 November 2022 / Published online: 16 December 2022

Abstract
Low-code platforms aim at allowing non-experts to develop complex systems and knowledgeable developers to improve their productivity in orders of magnitude. The greater gain comes from using components developed by experts capturing common patterns across all layers of the application, from the user interface to the data layer and integration with external systems. Often, cloning sample code fragments is the only alternative in such scenarios, requiring extensive adaptation to reach the intended use. Such customization activities require deep knowledge outside of the comfort zone of low code. To effectively speed up the reuse, composition, and adaptation of pre-defined components, low-code platforms need to provide safe and easy-to-use language mechanisms. This paper introduces `OSTRICH`, a strongly typed rich templating language for a low-code platform (OutSystems) that builds on metamodel annotations and allows the correct instantiation of templates. We conservatively extend the existing metamodel and ensure that the resulting code is always well-formed. The results we present include a novel type safety verification of template definitions, and template arguments, providing model consistency across application layers. We implemented this template language in a prototype of the OutSystems platform and ported nine of the top ten most used sample code fragments, thus improving the reuse of professionally designed components.

Keywords Metamodel templating · Typechecking templates · Parameter constraints · Low-code · Development productivity · Model reuse

1 Introduction

The productivity of the development process is a key driver of the software industry. Productivity metrics include multiple criteria, from the time used to produce the minimal viable product to the adaptability to change in maintenance activities to the correctness of the final result. Low-code platforms, like the OutSystems platform [18–20,27], aim at shielding the developer from the complexity of the software construction process. The OutSystems platform is a visual model-driven development and delivery platform that allows developers to create enterprise-grade web and mobile applications. OutSystems' customers use Service Studio, the platform's IDE, to design all the aspects of their applications in a single place, including user interface, business logic, database model, and integration with external systems. The platform provides type-safe, domain-specific languages (DSL) for all of these aspects.

Beyond lowering the complexity provided by using DSLs, another major factor in increasing productivity is the safe reuse of abstract code artefacts [16,24]. Code reuse is a well-known practice in software construction in general and essential in any healthy development process [17,22]. The core factor here is the use of abstraction layers, leading to parametrization, isolation, and information hiding. To this end, OutSystems provides some basic scaffolding mechanisms for the most common development patterns and sample screen templates that can be cloned and adapted. Such mechanisms accelerate the construction of repetitive and complex code patterns and sophisticated user interfaces designed by experts for the low-skilled or design-impaired developers. However, if pre-prepared samples work as an attractor for new users, the need for extensions is frequent, and the lack of success in replicating and adapting sophisticated code is usually a strong detractor.

Communicated by Shiva Nejati and Daniel Varro.

✉ João Costa Seco
joao.seco@fct.unl.pt

[1] OutSystems, Lisbon, Portugal

[2] NOVA LINCS and NOVA University Lisbon, Lisbon, Portugal

In this paper, we present an extended version of [18] by introducing (type) dependencies between template parameters. We address the lack of abstraction and parametrization mechanisms in the OutSystems model-driven approach and further explore the OSTRICH language, a template language for OutSystems applications. The terminology "templates", in low-code platforms and web development in general, does not fully correspond to the technical concept of parametrized code templates but to the weaker notion that includes cloning and in-place modification of samples. The use of such sample "Screen templates" is a common practice in the OutSystems platform [26] and other low-code platforms [23]. An example of a sample screen template is a page that is pre-prepared for listing *Products*, which after instantiation needs to be adapted to the actual database entity that the developer wants to use. The existing OutSystems screen templates are the direct motivation in the design of OSTRICH, but its application is not limited to UI. At its core, OSTRICH conservatively abstracts and parametrizes any element of the OutSystems metamodel.

The challenge we tackle with this paper is how to reuse pre-assembled applications with strong safety guarantees. We aim at having a template instantiation mechanism that produces well-formed code upon the validation of the arguments used to ground its parameters. Creating a screen from scratch, based on a sophisticated design, is cumbersome and can take a long time to get right. Cloning screen templates and ad hoc adaptation of code does solve this problem. However, adaptation requires deep knowledge of each template internals, thus not suitable for all levels of expertise. Our goal is to remove the need for ad hoc adaptation in the use of templates and replace it with the smart shaping of components to a set of contextualized arguments.

Our goals include having fully functional applications right after instantiation, with all arguments and encoded adaptations in place. Template instantiation should be a plug-and-play action in the IDE with immediate effect on the current application and a zero-cost abstraction (no overhead at runtime).

We adopt a model-driven approach and conservatively extend the OutSystems metamodel, which allows for seamless integration in the existing IDE, for both the construction and the instantiation of templates. Then, we develop a type-safe model that captures type-level computations, capable of producing new datatypes and model instances based on the structure of types and values given as arguments of the instantiation action. The produced model includes not only the structure of elements and their property values (i.e. the title of a screen), but also expressions defined during instantiation.

Our approach is type-safe, hence the verification of template code at design-time guarantees that all instantiations also produce valid code, upon the verification of the arguments. We implement a prototype that includes a typed language for expressions respecting the phase distinction [4], thus avoiding dependencies between compile-time and runtime expressions. We explain with greater detail, when comparing with [18], how the staged verification of expression is performed and how expressions are decomposed into compile-time and runtime fragments.

We evaluate our approach by analysing and adapting the set of the top ten most used, production-ready, screen templates from the OutSystems platform. We successfully converted nine out of ten onto OSTRICH templates, thus moving the adaptation time required after instantiating a screen template from a few hours to no time. Screen templates are widely used in the bootstrapping of applications, which elevates the impact of this work.

This paper is part of the research efforts of project GOLEM[1] that addresses automation of programming like synthesis and metaprogramming. The template language OSTRICH is instrumental in order to allow assemblies of larger components that can be matched to programming concepts, cf. [30].

To summarize, our contributions are as follows:

- A novel conservative extension of a low-code model to templates, in a market leader low-code platform, OutSystems. We allow the creation and edition of templates in the OutSystems IDE. We introduce it with an example (Sect. 2) and present its metamodel (Sect. 3). The support for type dependencies introduced in this paper expands the expressivity of OSTRICH in relation to the original article [18] and allows for more cases to be safely captured.
- A semantics for the model, via an instantiation algorithm (Sect. 4), and for the enclosed expression language with embedded compile-time computations (Sect. 5).
- A staged typing algorithm for template expressions (new with relation to [18]) that ensures that templates instantiated with valid arguments always yield valid runtime models (Sect. 5.2). We introduce a lightweight type dependency that solves a limitation previously identified which prevented developers from relating parameters in templates.
- An experimental evaluation, obtained by porting sample screens used in cloning and rudimentary adaptation mechanisms native in the OutSystems platform (Sect. 6).
- A critical review of template languages and models for programming languages and models (Sect. 7).
- A set of clear extension points and supporting features for more abstraction, adaptation, verification, and evolution mechanisms (Sect. 9).

[1] GOLEM: Automated Programming to Revolutionize App Development, A CMU|Portugal large scale research project, Ref. Lisboa-01-0247-Feder-045917.

Finally, we close the paper with some final remarks.

2 Templates by example

In this section, we use a running example to intuitively introduce OSTRICH, our template language. The language's metamodel is formally presented and discussed in Sect. 3.

OutSystems models follow a strict hierarchical structure to represent the *has* relationship: for each object in the model we identify the set of its *children* elements. Objects can also *use* other objects with no restrictions. For the sake of simplicity, in this paper, we support only the definition of applications consisting of entities (database tables) and screens. We define screens as containing a tree of user interface widgets that can depend on database tables to display data.

In Fig. 1, we depict an OutSystems application in the OutSystems IDE. In Fig. 2, we show its model. The application consists of entity `Product` and screen `ListProduct`. The entity has two attributes: `Description` of type String and `IsInStock` of type Bool. The screen contains a Table widget with a data dependency to entity `Product`. The table contains two Column widgets, one for each of the entity's attributes. Both columns have a ToggleVisibility widget to show/hide the corresponding column. The value of the `Description` attribute is displayed using a generic Value widget. For the `IsInStock` attribute, we use an Icon widget whose visibility is determined by the attribute's value.

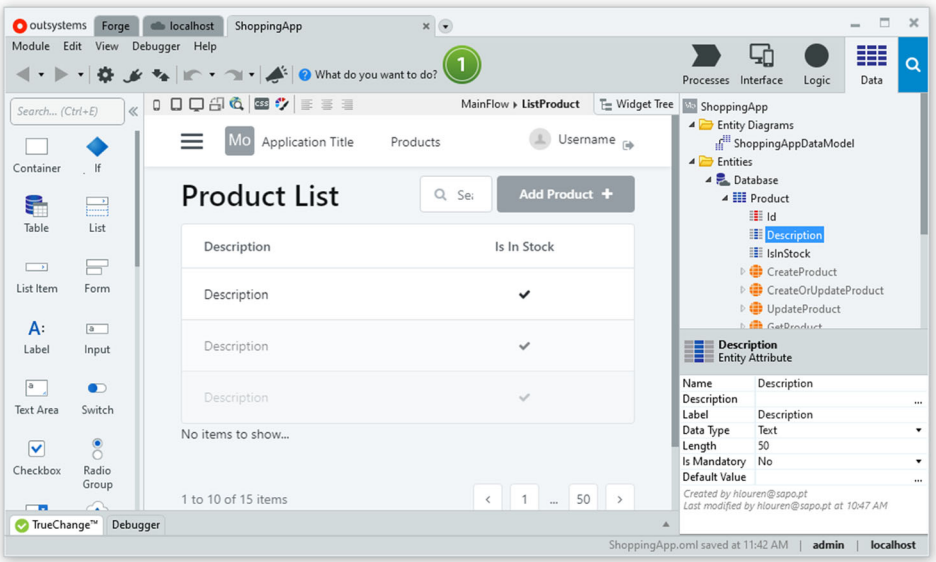
This pattern, a screen used for listing the content of a database table, is frequent enough that we might want to abstract it into a reusable template parametrized by an entity and the list of attributes of that entity to be displayed. The template may, for instance, include an elaborate design that one wants to propagate uniformly throughout the application. Such template can be modelled in OSTRICH as depicted in Fig. 3, where annotations nodes are conservatively added (in yellow) to a regular model. This template was defined using a modified version of the OutSystems IDE with support for OSTRICH (Fig. 4).

Under our approach, a template consists of a parametrized annotated model that, when removing the highlighted elements in Fig. 3, results in a *base* model that can be inspected and created using an unmodified version of our IDE. In order to ensure a well-formed model, our example template defines its own entity, `Sample`. This entity is used, for instance, as the source for the Table widget. When instantiating the template, we will want to effectively replace this entity with an actual entity provided by the developer. For that purpose, the template defines a parameter named `e` of type Entity(N,R). We also want to allow to specify the list of attributes of the entity to be displayed in the screen, instead of defaulting to showing all of the attributes. This is accomplished via the `attrs` parameter of type List<Attribute(N,T)>. Note the type dependency between the two parameters, which enforces the restriction that the attributes belong to the entity. This type dependency mechanism is explained in detail in Sect. 5.2.

Besides the declaration of parameters, OSTRICH supports the following annotations:

– *Property Value*: provides the value for a given property using a template expression. For instance, the Property Value annotation `t2` for screen `s` specifies that instead of the default name, `List`, the actual name is to be computed using the name of the entity provided through template parameter `e`. Properties that have not been annotated keep the value defined in the base model.

Fig. 1 Product list application



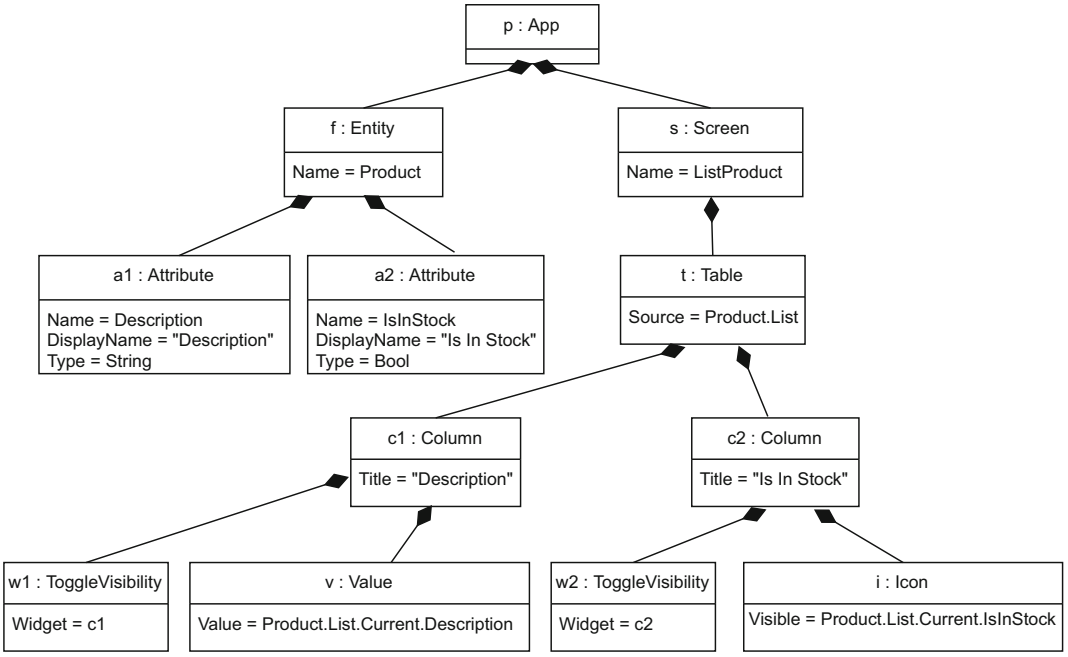


Fig. 2 Product list application model

– *Iteration*: repeats an element multiple times by iterating a compile-time list. The list is specified by a template expression, and a cursor name is introduced so that the list items can be referred to in the annotations of the element's children nodes. For instance, the iteration annotation `t4` in column `c` specifies that we must repeat the column once for each of the attributes of the entity provided through template parameter `e`. The cursor name `attr` refers to the attribute being iterated.
– *Conditional*: conditionally include or exclude an element. For instance, the conditional annotation `t6` in icon `i` specifies that this element is kept only when attribute `attr` is of type Bool.

A template is instantiated by providing a location in a target app and concrete values for the template parameters. The target location specifies the *insertion point* for the result of evaluating the template. In our example, we can easily see that the model in Fig. 2 is obtained by instantiating the template from Fig. 3 in the target app of Fig. 5, using App `p` as the target location and entity `Product` as the template's argument.

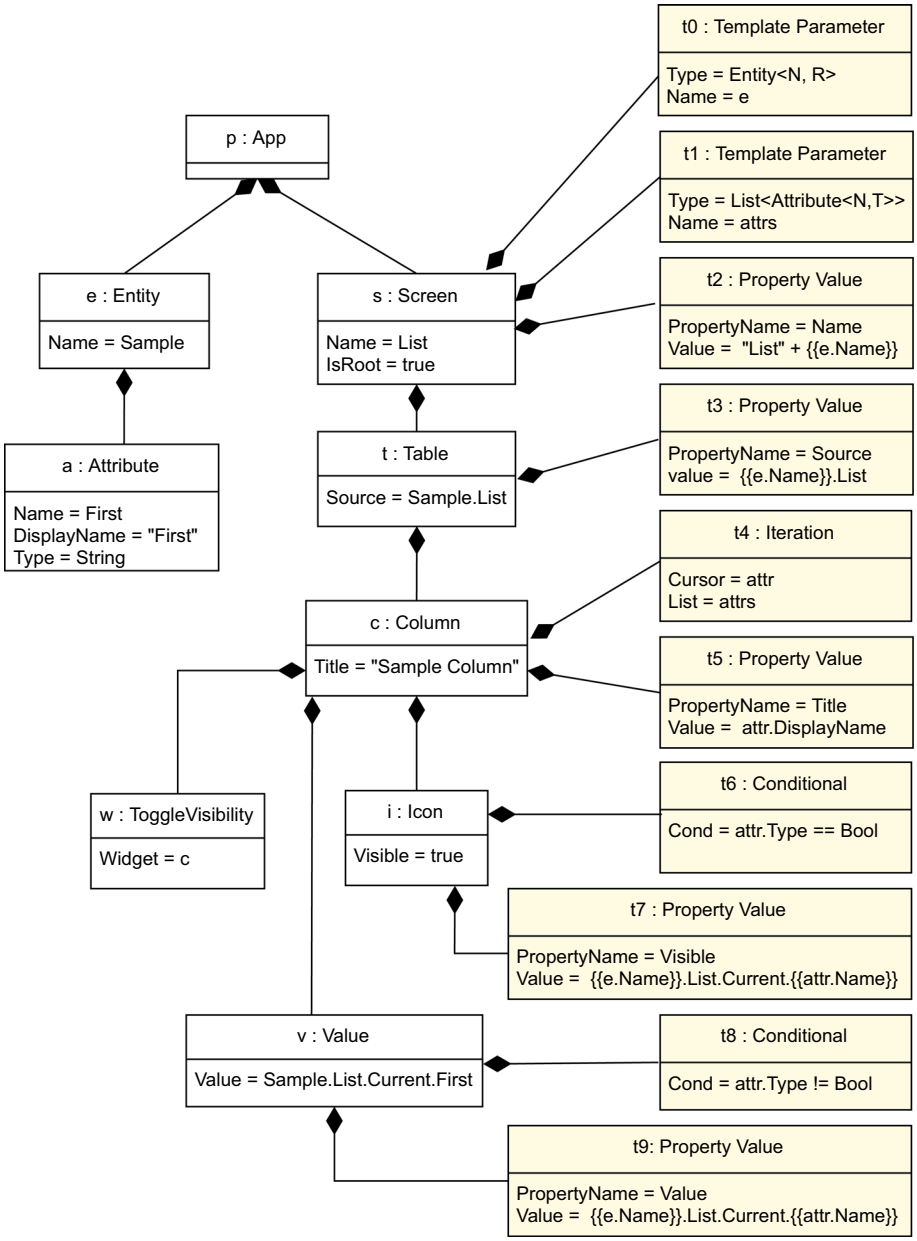
Comparing the example of Fig. 3 with the one presented in [18] highlights a substantial difference in terms of the expressiveness. The example template in this paper contains two inter-dependent parameters: an entity and a set of attributes of that entity. The previous version of `OSTRICH` was unable to express dependencies between parameters.

In this model, we present the basic abstraction mechanism by conservatively extending the base model of OutSystems applications. Hence, `OSTRICH` does not break existing code, and application models that do not use `OSTRICH` are completely forward compatible. Template edition and instantiation do require a production-ready ServiceStudio of our prototype, but the resulting models from instantiating a template are fully compatible with existing code and tools. Immediate improvements to the present approach are to include template instantiation as a language primitive. This extension requires new nodes in the metamodel and would break the backward compatibility of models using templates with the previous versions and tools, but allows for more modular structures of components. The compatibility of models with tool versions is already a concern with other features and is properly illustrated in prior work [20]. In this case, a primitive for instantiation templates within templates allows for different kinds of columns to be modularly captured in separate sub-templates for columns only to be instantiated in a table template. We present a basic model of type-level computation, where we can produce new data types and use them in newly created code. One crucial example is the creation of entities (which are datatypes in this language) in automatic synchronization processes that are defined from the arguments of a template.

3 Template metamodel

Figure 6 presents the underlying metamodel of the `OSTRICH` language, which builds on the metamodel of OutSystems applications. In this figure, uncoloured elements correspond

Fig. 3 List template model



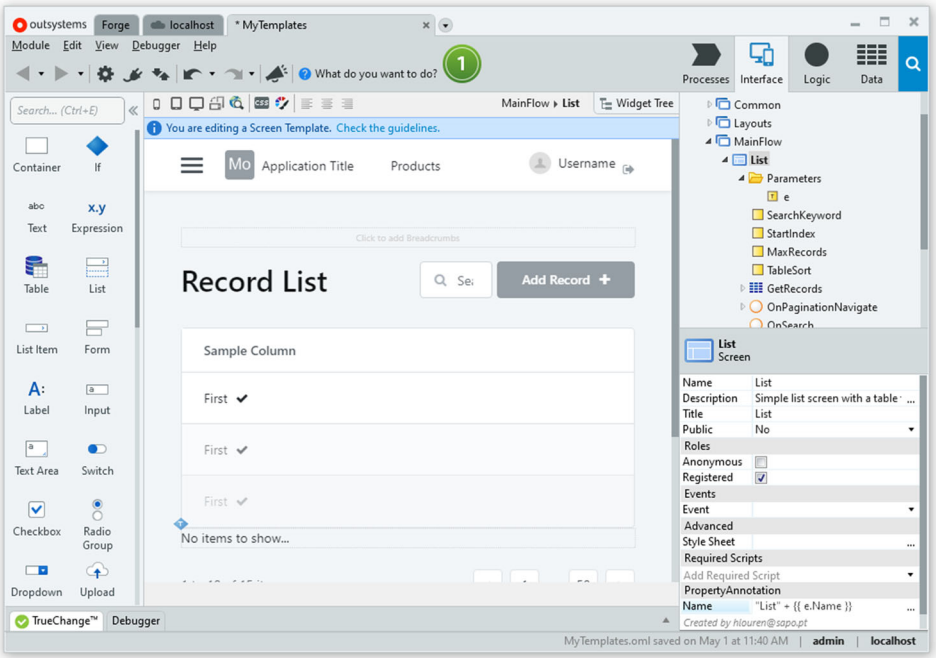
to (a simplified version of) the metamodel for OutSystems applications. The coloured elements are the ones that were introduced specifically by OSTRICH to support the definition of templates and template components.

Briefly, applications are composed of 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 that were omitted here for the sake of simplicity. Only the nodes used in the example were included in this metamodel. Abstract Object nodes *contain* other nodes, forming a parent-child hierarchical tree structure like the one we have for entities and attributes. Abstract Object nodes can also *use* other nodes. For instance, widget *ToggleVisibility* uses `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: provide instantiation-time values.
- Property annotation: set the value of a property dynamically at instantiation time.
- Iteration annotation: repeat an element once for each item in the provided list.
- Conditional annotation: dynamically include or exclude an element, depending on the condition value.

Fig. 4 List template



Such 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, makes it easy to extend the tools that need to take advantage of the annotations. That is the case of the OutSystems IDE, which was modified to allow editing template annotations. In Fig. 4, we can see a template being edited. Notice the *Property Annotation* section at the bottom right corner, where the value for the *Name* property annotation is set.

Nevertheless, this is not the only possible solution. The support for templates could be achieved in multiple ways. A possible alternative would be to create a template representation for each element of the original metamodel (e.g. screen and screen template, or table and table template). The obvious disadvantages include that the creation of these templates requires the costly extension of the existing tools.

4 Model semantics

In this section, we present our template instantiation algorithm and thus illustrate the semantics of the model. In our prototype, the instantiation of a template is available through an operation of the IDE. The inputs for the algorithm are: the template root object, the target parent object where the template is to be instantiated, and a compile-time value for each template parameter. The compile-time values range from basic value literals to model objects and lists of model objects. Figure 2 depicts the result of instantiating the template screen of Fig. 3 with the following inputs:

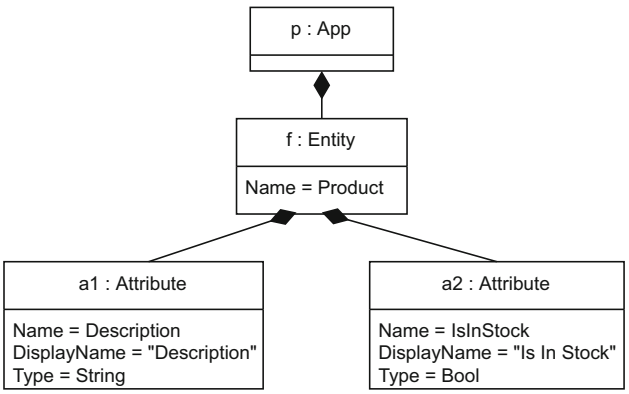


Fig. 5 Target application

- template: Screen `s` (Fig. 3)
- targetParent: App `p` (Fig. 5)
- arguments: { `t1` ↦ `f` }, which means that the template parameter `t1` is assigned with entity `f` of the caller context

Algorithm 1 starts by creating the environment `env` containing all the arguments of the instantiation (line 2) and the empty map `newObjs` to store the objects created during the instantiation process (line 3). The algorithm then proceeds in two steps. We recursively traverse the template, first, to create all structural objects in the target parent object, and then to evaluate all template expressions and set the property values on the newly created objects. This ensures that we properly resolve the object names and corresponding property values regardless of the order by which objects are created. Func-

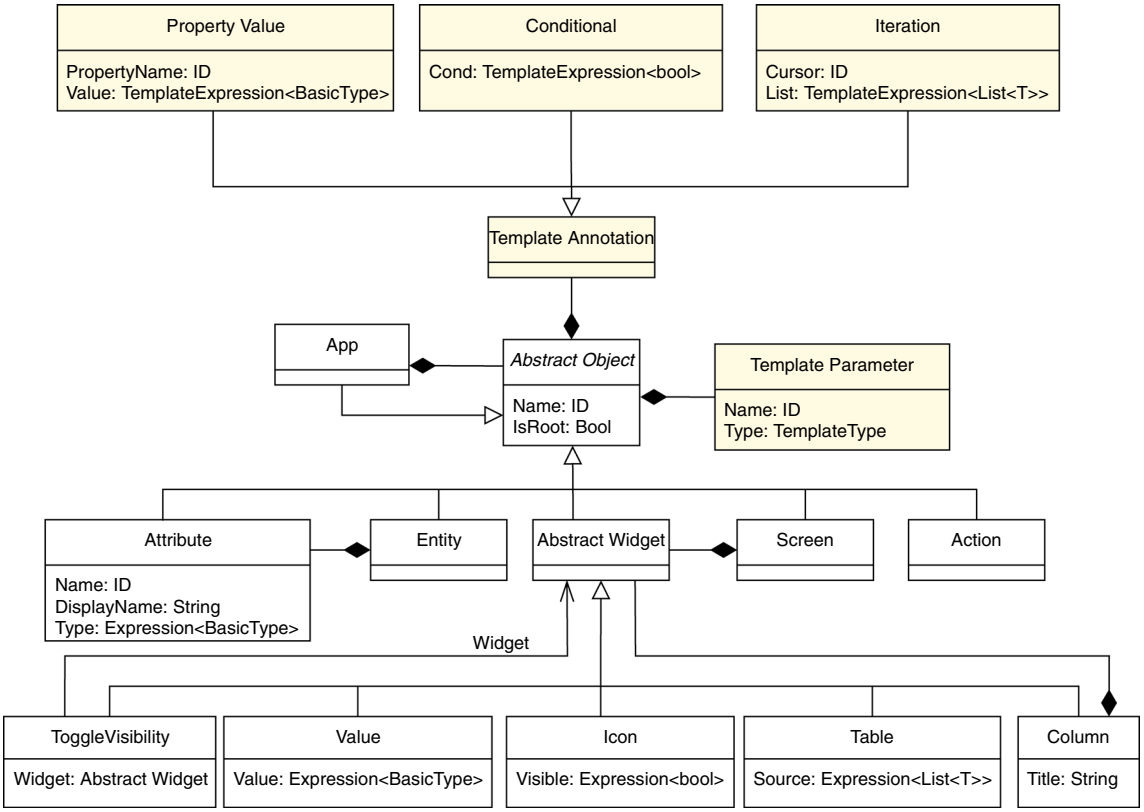


Fig. 6 Template metamodel

tion EVALUATEPROPERTY (line 32), used to evaluate template expressions, is discussed in detail in Sect. 5.

The TRAVERSE function (line 6) evaluates all conditional and iteration annotations present at each template node. For the case of conditional annotations (line 7) this amounts to evaluating the annotation's condition and skipping the node if the condition evaluates to `false`. In the case of annotations (line 11) it evaluates and iterates the annotation's list. Each iteration is evaluated in a newly created environment that binds the cursor name to the list item being iterated (line 15). This makes the item's value available for template expressions contained in the template node and its children. Notice that function TRAVERSE is parametric in function `fun`, which is called for all template nodes.

The CREATEOBJECT function (line 20), used in the first traversal of the model, creates new objects under the target parent node. Each new object is stored in the `newObjs` map. Notice that this is not a direct object-to-object map. Because of the loop in the TRAVERSE function, a single object in a template may correspond to multiple objects in the target app. For example, the iteration annotation `t4` in Fig. 3 will result in multiple instances of widgets `c`, `w`, `v`, and `i`. The `newObjs` map handles the multiplicity by using the pair `(templNode, getCursorsState(env))` as the map's key. A `CursorsState` value encapsulates the state of each cursor at a specific point in the instantiation process. In our example, we only have one iteration annotation `t4` whose list is `e.Attributes`, and thus the cursor state sequentially corresponds to each of the attributes of entity `Product`. That is, after running the first step of the instantiation algorithm (line 4) `newObjs` contains the following:

$$\begin{aligned}\texttt{newObjs} = \{&(s, []) \mapsto s, (t, []) \mapsto t,\\ &(c, [a1]) \mapsto c1, (w, [a1]) \mapsto w1, (v, [a1]) \mapsto v,\\ &(c, [a2]) \mapsto c2, (w, [a2]) \mapsto w2, (i, [a2]) \mapsto i\}\end{aligned}$$

We can see that e.g. column `c` in the template has two corresponding entries in `newObjs`, namely $(c, [a1]) \mapsto c1$ and $(c, [a2]) \mapsto c2$. This is a consequence of the list of iteration annotation `t1` being evaluated to `{a1, a2}`, which resulted in two new columns `{c1, c2}` being created.

Finally, function SETPROPERTIES is used in the second traversal (line 5) to set the property values for the newly created objects. This function starts by looking up the target object in `newObjs` and iterates the object's properties in order to set them. The actual property value is obtained by calling function EVALUATEPROPERTY (line 32). Property annotations, if they exist, are evaluated to determine the actual property value. If not, we use the property value from the template object as the default value. As a special case, if the value is a model object, then we try to use

Algorithm 1 Template instantiation algorithm (Part 1 of 2)

```
    input
      template: AbstractObject                                   ▷
    root template object with annotations
      parent: AbstractObject                    ▷ target parent object
      args: TemplateParameter → Object                           ▷
    template parameter mapping
1:  function instantiate(template, parent, args)
2:    env ← newEnv(args)
3:    newObjs ← {}
4:    TRAVERSE(template, parent, env, newObjs, createObject)
5:    TRAVERSE(template, parent, env, newObjs, setProperties)

    input
      templNode: AbstractObject                  ▷ current template object
      targetParent: AbstractObject               ▷ current target parent object
      env: ID → AbstractObject                   ▷ evaluation environment
      newObjs: AbstractObject × CursorsState → AbstractObject
            ▷ new objects indexed by (template object, cursor values)
      fun: {CREATEOBJECT, SETPROPERTIES}         ▷ traversal function
6:  function traverse(templNode, targetParent, env, newObjs, fun)
7:    if hasConditionalAnnotation(templNode) then
8:      if evaluate(getCondExpression(templNode), env) == true
    then
9:        fun(templNode, targetParent, env, newObjs)
10:     else skip
11:   else if hasIterationAnnotation(templNode) then
12:     list ← evaluate(getListExpression(templNode), env)
13:     cursorName ← getCursor(templNode)
14:     for all item in list do
15:       env ← beginScope(env)
16:       bind(env, cursorName, item)
17:       fun(templNode, targetParent, env, newObjs)
18:       env ← endScope(env)
19:   else fun(templNode, targetParent, env, newObjs)
```

Algorithm 2 Template instantiation algorithm (Part 2 of 2)

```
20: function createObject(templNode, targetParent, env, newObjs)
21:   obj ← createChild(targetParent, typeof templNode)
22:   newObjs ← newObjs ∪ {(templNode, getCursorsState(env)) ↦
    obj}
23:   for all child in getChildren(templNode) do
24:     TRAVERSE(child, obj, env, newObjs, createObject)

25: function setProperties(templNode, targetParent, env, newObjs)
26:   obj ← get(newObjs, (templNode, getCursorsState(env)))
27:   for all prop in getProperties(templNode) do
28:     value ← evaluateProperty(templNode, prop, env, newObjs)
29:     setPropertyValue(obj, prop, value)
30:   for all child in getChildren(templNode) do
31:     TRAVERSE(child, obj, env, newObjs, setProperties)

    input
      prop: Property                          ▷ property to be evaluated
32: function evaluateProperty(templNode, prop, env, newObjs)
33:   if hasPropertyAnnotation(templNode, prop) then
34:     return evaluate(getValueExpression(templNode, prop), env)
35:   else
36:     value ← getPropertyValue(templNode, prop)
37:     if contains(newObjs, (value, getCursorsState(env)) then
38:       value ← get(newObjs, (value, getCursorsState(env))
39:     return value
```

`newObjs` to map it. For example, consider ToggleVisibility widget `w` in Fig. 3. The value for its `Widget` property is `c`. The `newObjs` map presented above tells us that `c` must be mapped to `c1` and `c2` when the cursor value is `a1` and `a2`, respectively (Fig. 2). We need a two-pass algorithm to evaluate cross-references between objects without being dependent on the objects creation order: when mapping object `c` we need the new objects to have already been created. A single-pass algorithm cannot guarantee this in general.

One particularly useful aspect of templating is its compositionality, i.e. the instantiation of templates as a model element. This feature is out of the scope of the present work.

The next section describes the semantics of the template expressions used in the annotations. These expressions include not only the compile-time computation of literal values, but also the construction of runtime expressions that will use runtime values of the existing model elements.

5 Template expressions

The example of Sect. 2 illustrates the definition and application of templates in our model. As seen before, the operational semantics of the language, expressed in Algorithm 1, covers the interpretation of model annotations over the actual model nodes. The algorithm also includes the verification and evaluation of expressions within such annotations. We now explore the evaluation and verification algorithm for template expressions that ensures that all produced runtime expressions are well-formed.

The template expression language is a multi-stage language. The language defines a compile-time stage of evaluation whose results are runtime expressions, which is a strict subset of the template language and matches the target model of the OutSystems platform. The language draws the boundaries between type-level and compile-time computations and the runtime computations of the application. The typing discipline involving the template language guarantees the preservation of phase distinction [4] in the language, thus ensuring that all template instantiations are not dependent on runtime values and therefore essential for the termination of compile-time computations. Template expressions evaluate, in the first stage, using an environment containing references to the elements in the actual model, the parameters introduced in the template declarations, and the cursor values introduced by iteration annotations in the template. The resulting runtime expressions include references to runtime elements like entities, attributes, and aggregates. Such elements are usually the anchors to access the program state. We perform

the second-stage evaluation in a closed setting, where all abstracted references to model elements are inhabited by actual elements in the current model, and all the expressions associated with element properties are completely resolved to runtime values.

We now use a running example to informally introduce the syntax and semantics of template and runtime expressions. Consider the template expression used in the property box of List template (see Fig. 4), and present in the template model (Property Value annotation `t2` shown in Fig. 3),

```
"List"+{{e . Name}}
```
(1)

where `e` is a template parameter of type Entity. In this particular case, expression (1) is used to customize the screen name based on the name of the entity provided when instantiating the template. The concrete syntax of expression (1) uses the handlebars notation, common in well-known templating and embedded markup languages, to separate the stages of expressions. The abstract representation of expression (1), depicted in Fig. 7, is as follows:

```
"List" + (NameOf e)
```
(2)

We design the concrete syntax of the expression language to have embedded template language snippets. However, the "dot" operator inside the handlebars is context-dependent and is translated to the native operator (`NameOf _`) during the parsing process. We show later that, outside of the handlebars, the selection operation is translated to a runtime selection ($\hat{.}$).

Also, the comparison between values and types (`attr.type !=Bool`) is captured by the parser and translated to an abstract expression, of the form `not` (M `isOfType` T)[2].

Consider the example of an instantiation for the free variable e in expression (2) with entity **Product**, which has for the name property the identifier **Product**, we obtain the template expression

```
"List" + Product
```
(3)

that results, at compile-time, in a new name for a screen, **ListProduct**. The evaluation of expression (3) is possible because we overload the concatenation operator (+) to join a String literal with an existing name to define a new name. We show in Sect. 5.1 the detailed evaluation steps for this template expression.

We first describe some details of the abstract syntax of the language, defined by the grammar in Fig. 7. The language encompasses a sub-language of value literals with lists and records. For the sake of simplicity, constructors are only available for values. We adopt a compact presentation that uses the operator $\oplus$ in the abstract syntax to represent operations over basic values such as arithmetic operators, relational and Boolean operators, and concatenation on strings, the indexing on lists, and the selection operation on records. The access to particular properties of model elements like *Name*, *Label*, and *Type*, is defined by built-in language operations. In sync with this operator, $\hat{\oplus}$ represents the delayed form of the $\oplus$ operator. The delayed operations are left uninterpreted in the first stage of evaluation, which resembles staged computation approaches like [10,31].

The second stage of evaluation defines the domain of runtime expressions given by the syntax of Fig. 8. Notice that the field selection expression is now instantiated with a label instead of a term.

We also include references to model elements as language values (V) so that we can access their properties, as illustrated by template expression

```
attr . DisplayName
```
(4)

in annotation `t5` of Fig. 3. For brevity, from all elements in the OutSystems language, we only include entities and attributes. An entity template value is defined by a name, and a record type defining the name, and type of each attribute. An attribute template value is defined by the name of the entity it belongs to, a label, a type for that particular attribute, and a set of properties that can be used in expressions.

The first stage of template expressions evaluation is expressed by function $[\![M]\!]_{Env}$, defined in Fig. 9, and function $(\!|M|\!)$, omitted from this presentation and representing the straightforward interpretation of operations on literal values. For instance, it defines the selection of fields of records, concatenation of names, and other destructors of values. Function $[\![M]\!]_{Env}$ is defined with relation to an environment (Env) that defines a substitution for free variables in terms. We also use the auxiliary definitions shown in Fig. 10. Notice in Fig. 9 that $\oplus$ operations are fully interpreted in the first stage, by recursively interpreting the operands to values and immediately computing the result, an expression (E) of the language, also defined in Fig. 7. In the case of the operators $\hat{\oplus}$, the resulting operation is an uninterpreted binary operation on two terms. We will next show how to verify the soundness of a template expression. The invariant we aim to extract from such a verification procedure is the guarantee that well-typed template expressions always evaluate to valid runtime expressions. However, we analyse first an example of evaluation.

5.1 An example of evaluating a template expression

To better explain the staged semantics of expressions, we will use the following template expression

```
{{e . Name}} . List . Current . {{attr . Name}}
```
(5)

[2] The unary operator `not` is actually encoded in the abstract language using the binary operator `nand`.

$$
\begin{array}{lll}
v & ::= & \textbf{(values)} \\
& | \quad num & \text{(number literal)} \\
& | \quad string & \text{(text literal)} \\
& | \quad bool & \text{(boolean literal)} \\
& | \quad [\overline{v}] & \text{(list literal)} \\
& | \quad \{\overline{L = v}\} & \text{(record literal)} \\
V & ::= & \textbf{(model elements)} \\
& | \quad \mathit{Entity}\langle N, T\rangle & \text{(entity element)} \\
& | \quad \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle & \text{(attribute element)} \\
M & ::= & \textbf{(terms)} \\
& | \quad x & \text{(variable)} \\
& | \quad L & \text{(labels)} \\
& | \quad N & \text{(name)} \\
& | \quad v & \text{(value literal)} \\
& | \quad V & \text{(model element)} \\
& | \quad M \oplus M & \text{(template operation)} \\
& | \quad M \mathbin{\hat{\oplus}} M & \text{(boxed operation)} \\
& | \quad \texttt{NameOf}\ M & \text{(name property)} \\
& | \quad \texttt{LabelOf}\ M & \text{(label property)} \\
& | \quad M\ \texttt{isOfType}\ T & \text{(type test)}
\end{array}
$$

Fig. 7 Syntax of template multi-stage expression language

$$
\begin{array}{lll}
E & ::= & \textbf{(expression)} \\
& | \quad v & \text{(value literal)} \\
& | \quad x & \text{(variable)} \\
& | \quad N & \text{(name)} \\
& | \quad E \oplus E & \text{(value operation)} \\
& | \quad E \,.\, L & \text{(field selection)}
\end{array}
$$

Fig. 8 Syntax of the target runtime expression language

$$
\begin{array}{ll}
[\![v]\!]_{Env} = v & \text{(E-Val)} \\
[\![x]\!]_{Env} = Env(x) & \text{(E-Var)} \\
[\![L]\!]_{Env} = L & \text{(E-Label)} \\
[\![N]\!]_{Env} = N & \text{(E-Name)} \\
[\![M \,.\, M']\!]_{Env} = (\!|\, (\texttt{PropsOf}\ V) \,.\, L \,|\!) & \text{(E-Field)} \\
\quad \text{with } [\![M]\!]_{Env} = V \text{ and } [\![M']\!]_{Env} = L & \\
[\![M \oplus M']\!]_{Env} = (\!|\, v \oplus v' \,|\!) & \text{(E-ValOp)} \\
\quad \text{with } [\![M]\!]_{Env} = v \text{ and } [\![M']\!]_{Env} = v' \text{ and } \oplus \neq \,. & \\
[\![M \mathbin{\hat{\,.}} M']\!]_{Env} = [\![M]\!]_{Env} \,.\, L \text{ with } [\![M']\!]_{Env} = L & \text{(E-Sel)} \\
[\![M \mathbin{\hat{\oplus}} M']\!]_{Env} = [\![M]\!]_{Env} \oplus [\![M']\!]_{Env} \text{ with } \hat{\oplus} \neq \mathbin{\hat{\,.}} & \text{(E-Exp)} \\
[\![\texttt{NameOf}\ M]\!]_{Env} = \texttt{NameOf}\ [\![M]\!]_{Env} & \text{(E-NameOf)} \\
[\![\texttt{LabelOf}\ M]\!]_{Env} = \texttt{LabelOf}\ [\![M]\!]_{Env} & \text{(E-LabelOf)} \\
[\![M\ \texttt{isOfType}\ T]\!]_{Env} = [\![M]\!]_{Env}\ \texttt{isOfType}\ T & \text{(E-TypeOf)}
\end{array}
$$

Fig. 9 Semantics of template expressions

$$
\begin{array}{l}
\texttt{NameOf}\ \mathit{Entity}\langle N, T\rangle \triangleq N \\
\texttt{LabelOf}\ \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle \triangleq L \\
\mathit{Entity}\langle N, T\rangle\ \texttt{isOfType}\ T \triangleq \mathit{True} \\
\mathit{Entity}\langle N, T\rangle\ \texttt{isOfType}\ T' \triangleq \mathit{False} \text{ with } T \neq T' \\
\mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle\ \texttt{isOfType}\ T \triangleq \mathit{True} \\
\mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle\ \texttt{isOfType}\ T' \triangleq \mathit{False} \text{ with } T \neq T' \\
\texttt{PropsOf}\ \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle \triangleq \{\overline{L' = v}\}
\end{array}
$$

Fig. 10 Inspection functions

that is used in an iteration context over the attributes of a parameter of kind entity. The template variables `e` and `attr` are used to refer to an entity and to one element of a list being iterated. The abstract representation of expression (5) in the grammar shown in Fig. 7 is the following

$$(((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}) \mathbin{\hat{\,.}} \mathit{Current}) \mathbin{\hat{\,.}} (\texttt{LabelOf}\ \mathit{attr}) \qquad (6)$$

Expression (6) will be evaluated with the following evaluation environment

$$
\begin{array}{l}
\mathit{Env} \triangleq [e = \mathit{Entity}(\textbf{Product}, \{\textbf{Description}: \mathit{String}, \\
\textbf{IsInStock}: \mathit{Bool}\}), \\
\qquad \mathit{attr} = \mathit{Attribute}(\textbf{Product}, \textbf{IsInStock}, \mathit{Bool}, \\
\qquad\qquad \{\textbf{DisplayName} = \texttt{"Is In Stock"}\})\,]
\end{array}
$$

where entity named **Product**, and attributes named **Description** and **IsInStock** are existing model elements. Moreover, the runtime environment includes built-in labels *List* and *Current*, used to access the list of records of an entity and the current record of a list being iterated, respectively. Thus, expression (6) is evaluated by recursively applying the rules shown in Fig. 9 as follows,

$$
\begin{array}{l}
[\![(((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}) \mathbin{\hat{\,.}} \mathit{Current}) \mathbin{\hat{\,.}} (\texttt{LabelOf}\ \mathit{attr})]\!]_{Env} = \\
[\![((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}) \mathbin{\hat{\,.}} \mathit{Current}]\!]_{Env} \,.\, [\![\texttt{LabelOf}\ \mathit{attr}]\!]_{Env} = \text{(E- Sel)} \\
[\![((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}) \mathbin{\hat{\,.}} \mathit{Current}]\!]_{Env} \,.\, \texttt{LabelOf}\ [\![\mathit{attr}]\!]_{Env} = \text{(E- LabelOf)} \\
[\![((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}) \mathbin{\hat{\,.}} \mathit{Current}]\!]_{Env} \,.\, \textbf{IsInStock} = \text{(E- Var)} \\
\quad \text{with } \mathit{Env}(\mathit{attr}) = \mathit{Attribute}(\textbf{Product}, \textbf{IsInStock}, \ldots) \\
[\![((\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List})]\!]_{Env} \,.\, [\![\mathit{Current}]\!]_{Env} \,.\, \textbf{IsInStock} = \text{(E- Sel)} \\
[\![(\texttt{NameOf}\ e) \mathbin{\hat{\,.}} \mathit{List}]\!]_{Env} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \text{(E- Label)} \\
[\![\texttt{NameOf}\ e]\!]_{Env} \,.\, [\![\mathit{List}]\!]_{Env} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \text{(E- Sel)} \\
[\![\texttt{NameOf}\ e]\!]_{Env} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \text{(E- Label)} \\
\texttt{NameOf}\ [\![e]\!]_{Env} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \text{(E- NameOf)} \\
\textbf{Product} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} \text{(E- Var)} \\
\quad \text{with } \mathit{Env}(e) = \mathit{Entity}(\textbf{Product}, \ldots)
\end{array}
$$

Notice that this result is a valid expression, in the language defined in Fig. 8, that can be assigned to the model element and evaluated at runtime (property Visible of Icon `i` depicted in Fig. 2).

Consider now the evaluation of template expression (2) used in annotation `t2` for the name property of the screen in

example 3:

$[\![\texttt{"List"} + (\texttt{NameOf}\, e)]\!]_{Env} =$
$(\!| [\![\texttt{"List"}]\!]_{Env} + [\![(\texttt{NameOf}\, e)]\!]_{Env} |\!) =$ (E- ValOp)
$(\!| \texttt{"List"} + [\![(\texttt{NameOf}\, e)]\!]_{Env} |\!) =$ (E- Val)
$(\!| \texttt{"List"} + (\texttt{NameOf}\, [\![e]\!]_{Env}) |\!) =$ (E- NameOf)
$(\!| \texttt{"List"} + \textbf{Product} |\!) =$ (E- Var)
with $Env(e) = Entity(\textbf{Product}, \ldots)$
$(\!| \textbf{ListProduct} |\!) = \textbf{ListProduct}$(def. of + and $(\!| \, |\!)$ function)

Finally, the evaluation of expression (4) used in annotation `t5` for the title property of the column widget in example 3:

$[\![attr\,.\,\textbf{DisplayName}]\!]_{Env} =$
$(\!| (\texttt{PropsOf}\, V)\,.\,\textbf{DisplayName} |\!) =$ (E- Field)
with $V = [\![attr]\!]_{Env} = Env(attr) =$
$Attribute(\textbf{Product}, \textbf{IsInStock}, \ldots)$
and $[\![\textbf{DisplayName}]\!]_{Env} = \textbf{DisplayName}$
$(\!| \{\textbf{DisplayName} = \texttt{"Is In Stock"}\}\,.\,\textbf{DisplayName} |\!) =$
$(\!| \texttt{"Is In Stock"} |\!) = \texttt{"Is In Stock"}$(def. of . and $(\!| \, |\!)$ function)

The examples above illustrate the semantics of the expressions within template property expressions and annotations. Some produce ground values to shape the instantiation or to be used as literals in the final model, others produce delayed runtime expressions, cf. expression (6). The well-formedness of the expressions, and of model templates thereof, is then established by a statically verified type system, discussed next.

5.2 Well-formedness of template expressions

A good development experience is highly dependent on the expressiveness and soundness of the frameworks' underlying language, in this case, the OSTRICH template model and its expression sublanguage. Ensuring the soundness of OSTRICH basically means that any instantiation of a template, which is a compile-time computation, using arguments that are compliant with the specification of the corresponding template parameters, must always produce a valid runtime model. This concern is not at the core of all the templating languages, especially the ones targeting user interfaces, because the target languages are very flexible and usually untyped. This is not the case when composing and generating program elements for a strongly typed language. In a low-code setting, it is not admissible to have to tweak the instantiated result to be able to execute without errors. By definition of low code, the programming environment leads the developer as much as possible through a path of valid program or application steps.

The metamodel and the expression language of OutSystems applications and the OSTRICH templates are strongly typed and the type system ensures that all template instantiations produce valid runtime expressions when using valid arguments.

$b ::= Num \mid Bool \mid String$		(Basic Types)
$t ::=$		(Types)
	b	(Basic Types)
$\mid$	$\{\overline{L : b}\}$	(Record Types)
$\mid$	$List(b)$	(List Types)
$\mid$	T	(Type Variable)
$\mid$	$\texttt{Label}(L)$	(Label Types)
$\mid$	$\texttt{Entity}(N, t)$	(Entity Types)
$\mid$	$\texttt{Attribute}(N, b)$	(Attribute Types)
$\mid$	$\texttt{LabelAttr}(N, b)$	(Label Types)
$\mid$	$\texttt{RecordAttr}(N)$	(Record Types)
$\mid$	$\texttt{BoxT}(t)$	(Delayed Types)

Fig. 11 Syntax of types

$M : t$	$M' : t'$	$\oplus/\hat{\oplus}$	$t \oplus t'$
$M : Num$	$M' : Num$	$+ \; - \; * \; /$	Num
$M : b$	$M' : b$	$= < \leq > \geq$	$Bool$
$M : Bool$	$M' : Bool$	`and or nand`	$Bool$
$M : String$	$M' : String$	`+`	$String$
$M : \{\overline{L : b}\}$	$M' : \texttt{Label}(L_i)$	`.`	b_i
$M : \{\overline{L : b}\}$	$M' : \texttt{Label}(L_i)$	$\hat{.}$	$\texttt{BoxT}(b_i)$
$M : \texttt{RecordAttr}(N)$	$M' : \texttt{LabelAttr}(N, b)$	$\hat{.}$	$\texttt{BoxT}(b)$
$M : List(b)$	$M' : Num$	$[\,]$	b

Fig. 12 Table for operator typing

In this section, we focus on the template expression language that captures the correct usage and correct typing of model element's names, and labels of element properties, and record and list typed expressions. The typing algorithm for the model fragment is a straightforward traversal and collection of declared names to define a typing environment for expressions. The template expression language is a staged language, cf. [10,31]. This means that there is a clear separation of phases [4] driven by the type system between the instantiation-time construction of expressions, and the execution of those expressions with runtime values. We use singletons as names to represent the dependencies between entities and their attributes, and ensure statically that all label usages are correctly checked as part of the target entity or record type. Similarly, the OutSystems model uses unique keys to track down such dependencies as well as changes in the model.

Figure 11 defines the type language associated with OSTRICH. Types describe the nature of the values of runtime computations (basic values b, records $\{\textbf{L : b}\}$, and lists $List(b)$) and values of compile-time computations (entities, their attributes, and labels of such attributes). The types

of entities (`Entity`(N, t)) contain a name N that statically identifies a singleton compile-time value and a type that can be specific of a type variable T. For the sake of language usability, names and variables are implicitly declared in the scope of a template by analysing the free names and type variables available. The type of attributes `Attribute`(N, b) identifies the entity to which an attribute belongs and its (basic) type. Types of labels `Label`(L) statically specify the label being selected. We also define types for labels of attributes that abstract the label being processed but fix the type of their values (`LabelAttr`(N, b)). This allows for a typesafe iteration over attributes of an entity. The same is defined for an abstract record type of an entity (`RecordAttr`(N)) that maintains the link to the source entity. This allows the typing of a selection operation, together with a label of type `LabelAttr`(N, b) without knowing the actual label being selected. Finally, we have type `BoxT`(t) describes compile-time expressions that denote delayed runtime expressions with type t. The use of this type is next illustrated by means of an example. Notice that the types used to describe attributes and corresponding labels share a name with the source entity. This lightweight type dependency allows us to abstractly check expressions within templates without knowing, *a priori*, the actual entity and its attributes. For instance, in Fig. 13, we define a template for columns that enforces that the attribute given as argument to parameter `attr` (t_2) belongs to the entity given as argument to parameter `e` (t_1). This example is similar to the one in Fig. 3, in which a template for a screen is defined. The `attrs` (t_1) parameter is a list of attributes that must be belong to the entity provided via parameter `e` (t_0). In both cases the dependency is established by name N, which is implicitly declared in the entity parameter and used in the attribute(s) parameter to bind them. With this binding we are able to abstractly check the expression in node t_3 of Fig. 13 and determine its type as `T` (also declared in template parameter `attr`).

We check all expressions with binary operators using the relation defined in Fig. 12. Notice that the delayed selection operator ($\hat{.}$) is also checked using this relation, and that the dependencies between the labels and the container values are properly checked using the name registered in their types. To allow genericity of entity and attribute values, names are abstracted in the definition of templates to be freely used to create dependencies between their parameters, cf. parametric polymorphism.

To better understand the typechecking process, we resort to an example to informally explain the process of staged typechecking used in our model. Recall the abstract expression 6 introduced in Sect. 5.1 representing the expression in concrete syntax in Fig. 13:

$$(((\texttt{NameOf}\ e)\ \hat{.}\ List)\ \hat{.}\ Current)\ \hat{.}\ (\texttt{LabelOf}\ attr)$$

Algorithm 3 Typechecking algorithm

```
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:     num ≙ Num
4:     bool ≙ Bool
5:     string ≙ String
6:     [M] | TYPEOF(M_i, c-env, r-env) = t_i ≙ [ t ]
7:     {L = M} | TYPEOF(M_i, c-env, r-env) = b_i ≙ {L : b}
8:     x | x : t ∈ c-env ≙ t
9:     x | x : t ∈ r-env ≙ BoxT(t)
10:     L | L : t ∈ c-env ≙ t
11:     L | L : t ∈ r-env ≙ BoxT(t)
12:     N | N : t ∈ c-env ≙ t
13:     M ⊕ M' ≙ ···          ▷ (see Figure 12)
14:     M ⊕̂ M' ≙ ···          ▷ (see Figure 12)
15:     NameOf(M) | TYPEOF(M, c-env, r-env) = Entity(N, t) ≙
16:       BoxT( {list : {current : RecordAttr(N)}} )
17:     LabelOf(M) | TYPEOF(M, c-env, r-env) = Attribute(N, b)
    ≙
18:       BoxT( LabelAttr(N, b) )
19:     M isOfType T | TYPEOF(M, c-env, r-env) = t ≙ Bool
20:   end
```

Notice two sub-expressions which are clearly compile-time (`NameOf` e and `LabelOf` $attr$) and that the selection operations are using the runtime/delayed selection expression ($\hat{.}$). To preserve the property of phase distinction, all compile-time subexpressions inside runtime expressions can use compile-time variables (e and $attr$) but must be rewritten as runtime expressions in the end. So, expressions `NameOf` e and `LabelOf` $attr$ need to be replaced by the result of its compile-time execution, a runtime value or expression. Our algorithm, presented in Algorithm 3, ensures that no compile-time computation depends on any value that is only obtained at runtime by structurally separating expressions and using two different environments to define identifiers: a compile-time environment to declare compile-time variables and a runtime environment to declare runtime variables. Hence, we can guarantee that all free variables occurring in runtime expressions only map to other runtime expressions.

That is what happens with expression 6: the variables denoting the results of the `NameOf` and `LabelOf` functions are declared in the runtime environment. The frontier between stages is inferred based on type information as we present here and not explicitly signalled as in the more general work of [10]. The two functions used above are declared with the following type signatures:

$$\texttt{NameOf}\ e : \texttt{Entity}(N, t) \rightarrow$$
$$\texttt{BoxT}(\{List : \{Current : \texttt{RecordAttr}(N)\}\})$$
$$\texttt{LabelOf}\ attr : \texttt{Attribute}(N, b) \rightarrow$$
$$\texttt{BoxT}(\texttt{LabelAttr}(N, b))$$

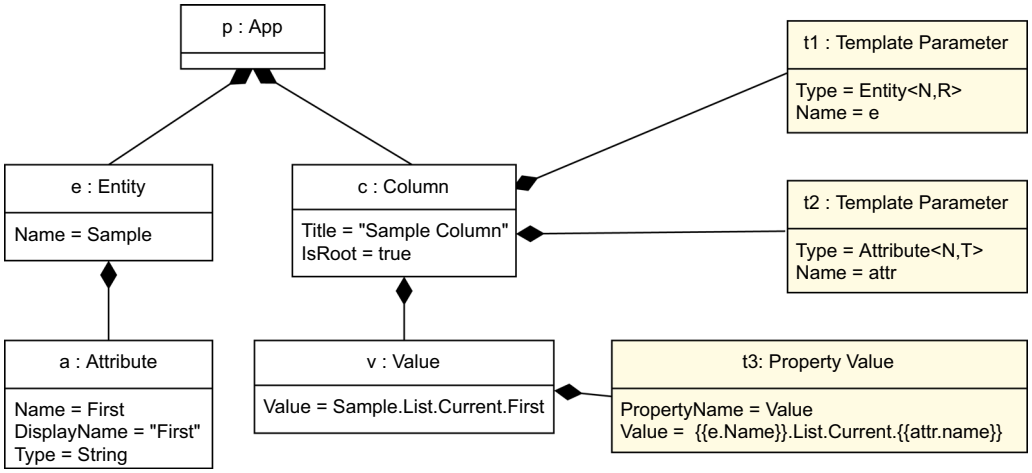


Fig. 13 Type dependency in template definition

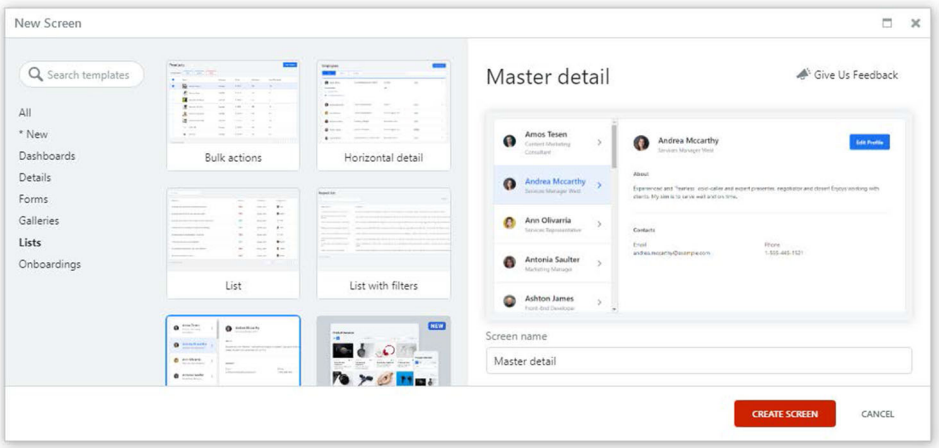


Fig. 14 New Screen dialog, showing the available Screen Templates

Recall that type `BoxT`(t) describes a runtime expression that produces a value of type t. The type of the result of `NameOf` e is a nested record that ultimately culminates in records that contain attributes from the entity *Product* (in the example of Sect. 5.1, expression 6). This intricate structure results directly from the structure of iterators in the OutSystems language and is not particular of our more open model. This is the information used to type the delayed operations $\hat{.}$. Function `LabelOf` $attr$ denotes the label of an attribute, thus conveying information about the attribute type and the entity to which it belongs, `LabelAttr`($Product$, $Bool$), thus allowing to check the selection operation without knowing exactly which attribute is being handled. The overall expression is valid if there is a dependency between the types of `NameOf` and `LabelOf`, by checking if $N = N'$, given `RecordAttr`(N) and `LabelAttr`(N', b). Thus, we prevent the selection of attributes that do not belong to that particular entity. The type of expression 6 is the boxed type of a runtime type extracted from the label type. In this example, assuming that the attribute is `IsInStock`, the type is `BoxT`($Bool$).

The compile-time evaluation represents the code generation phase in the OutSystems platform. At this stage, all ground template expressions are rewritten by resolving all compile-time expressions and replacing the delayed operations with their regular runtime counterparts.

6 Evaluation

OutSystems developers take advantage of a pre-defined set of sample code fragments, created by experts, to bootstrap and create their user interfaces and associated functionality. These code fragments are inappropriately called "screen templates" as they are complete and closed fragments of the application model. Thus, they still need to be fully customized after being cloned. Such a manual adaptation process necessarily creates errors and requires significant effort, programming skill, and deep knowledge of the tem-

plate's structure. One of the design goals for OSTRICH is to support the conversion of such screen templates into proper parametrized templates that require little to no customization after instantiation.

Currently OutSystems provides a set of 70 screen templates, organized into different categories such as List, Form, and Dashboard screens (cf. Fig. 14). To evaluate our work, we selected the top ten most instantiated screen templates (table 1) and converted them into OSTRICH templates. Collectively, this set of templates accounts for 53.9% of all screen template instantiations after three years of generalized use in the platform. All of the existing screen templates use predefined *sample* entities that the developers must replace with their *actual* entities. The initial step of our conversion process consisted of the definition of template parameters to allow developers to specify the core entities at template instantiation time.

Upon instantiation, all the elements in the screen that depend on sample entities or their attributes are assigned Property annotations to point to the corresponding attributes of the template parameter. Repeated or sequential uses of entity attributes such as entity fields in a form, or columns in a table, were decorated with iteration and conditional annotations to be iterated, used, or removed at instantiation time.

Some templates require additional parameters to shape their final result. For instance, *List with charts*, which shows a table and a pie chart, requires the attribute by which the data is to be grouped in order to define the pie chart categories.

Table 2 summarizes our results on the number of changes made when adapting each one of the screen templates. Notice the uniformity in the number of conditional annotations. This is not surprising since, in most cases, conditional annotations are used to select between alternative widgets based on the type of an attribute, and we have a fixed set of supported data types. We were able to successfully convert nine out of the ten screen templates considered. The only template that we were not able to convert is the *Account dashboard*. This template consists of a screen designed to display bank account information. It was not converted because it is highly specific, requiring constraints on the entity parameter that we currently cannot specify (the entity needs to contain a particular set of attributes, e.g. the account number). Most of the remaining 60 templates to be converted are not dependent on the structure of the incoming entities. Thus, it is highly foreseeable that those screen templates can be converted to our model in due time. We plan to address this kind of constraint and therefore increase the coverage of the supported templates.

Since we are in the prototyping phase and OSTRICH is not yet fully generally available, we were unable to conduct large-scale usability tests using OSTRICH. Internal usability tests define a baseline to measure the decrease in time when using OSTRICH templates, compared to cloning and adapting "screen templates". The time it takes to adapt "screen templates" ranges from 30 minutes to a few hours. A common scenario is when a user clones a "screen template", fails to adapt it to its current context, and gives up after a few erroneous trials. The advantage of OSTRICH, crucial in low-code platforms, is the absence of an error-prone adaptation phase.

The mechanisms proposed in this paper have been partially implemented in Service Studio and are currently being tested. Namely, the existing screen template mechanism was extended to support Boolean template parameters and If annotations. A new set of (now parametrized) screen templates has been created. When instantiating such a template, Service Studio will allow the developer to choose the parameter values.

Figure 15 illustrates the new functionality. In this figure, the developer is about to create a new screen based on the "Account Overview" template. For this particular instance, they have decided to include account movements (the "List Movements" checkbox is checked) but not balance information (the "Check Balance" checkbox is unchecked).

6.1 Limitations

While converting the screen templates, we identified in [18] a few limitations in the use of OSTRICH in more advanced scenarios. In this paper, we already proceeded to solve some of those limitations, namely the dependency between parameters.

At this stage, OSTRICH already allows for direct dependencies between parameters by means of singleton names as compile-time values. This allows us, as shown in the template of Fig. 3, to have a Table or a Form template with an entity and a list of attributes to be displayed as parameters. We are now able to check that the list of attributes only contains attributes that belong to the given entity.

Nevertheless, we can still identify room for improvements in this language and the verification mechanism of the model and expressions:

- It is not yet possible to identify a minimal set of attributes that an entity should have in order to be accepted as an argument for a template. The challenge of this task is to establish a universal subtyping relation between compile-time values, maintaining the dependencies between types. This limitation prevents us from checking the tenth example in our benchmark–the *Account dashboard* application.
- An improvement on the work presented in this paper is the extension of the lightweight type dependency to a more general type-dependent setting. This topic needs to be supported by extra evaluation work on the set of templates available in benchmarks and also new templates

developed with the current status of OSTRICH in new user studies.

- One limitation introduced by our design is the absence of a template instantiation node in the language. Although this would bring more expressiveness and reusability to the language, it would be incompatible with the current underlying model. So, any initiative to introduce this in the language has to be part of a larger effort in evolving the underlying model in the same lines.
- Lastly, template instantiation in OSTRICH is still a *write once* process. Template instantiation works by expanding the template definition in place and executing the template expressions in annotations in the target context to obtain runtime expressions that can be used in an application. After instantiation, the natural step is to continue editing the application, adding new elements, configuring the existing ones, or removing the ones that were added by the template. Once edited, changing the template applied or changing any of the arguments would override the customization made. Future developments of OSTRICH also include this aspect, with one strong requirement being to keep the soundness of the model being produced. The most promising approach is to keep the log of applied operations together with the template instantiation node. Then, when a change occurs, the type system can reevaluate all operations and discard or give a warning to the developer. This extension is also incompatible with the current status of the underlying model for OutSystems applications.

Table 1 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
Four column gallery	3.2
Account dashboard	3.2
Master detail	3.0
Total	53.9

7 Related work

Model to Model transformations

Templating, as we capture in this paper, is an abstraction mechanism added on top of the OutSystems language, cf. functions in general-purpose languages. Thus, templates exist, are edited and produce results at the same abstraction level as regular OutSystems code, and use the same development tools, thus dramatically increasing development efficiency.

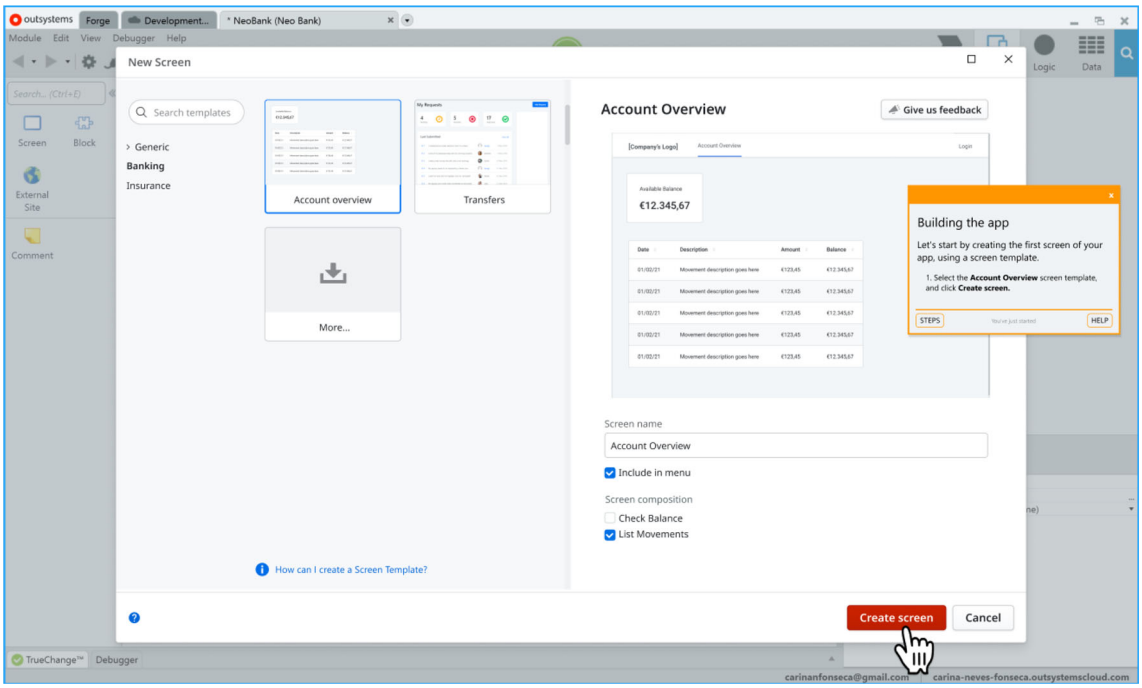


Fig. 15 New Screen dialog, showing a parameterized Screen Template

Table 2 Screen templates adaptation statistics

Screen template	Parameters	Annotations		
		Property	Iteration	Conditional
List with charts	2	15	2	6
Admin dashboard	2	20	3	6
Detail	1	24	2	3
Dashboard	2	20	3	6
List	1	17	2	6
List with filters	2	19	4	6
Bulk actions	1	17	2	6
Four column gallery	3	17	2	6
Account dashboard	–	–	–	–
Master detail	2	20	3	12
Total	16	169	23	57

Model to Model transformations (M2M) [9] in EMF [33] or MPS [14,29] define a process that is external to the language itself (cf. ATL [1]), usually defined by a set of rewriting rules or some low-level code that manipulates the syntax tree, and whose approach requires a skilled "language designer" to encode them.

We do define a kind of very generic M2M transformation, whose semantics is uniform and compositional. We argue that our approach cannot be encoded in any known M2M tool. Moreover, the constraints that our prototype checks that relate the nature of template parameters go well beyond syntactic soundness and usual integrity constraints in OCL. Such verifications are needed to ensure that our model is able to verify template code and produce well-typed programs in low code.

General purpose programming languages

Standard template languages take no advantage of the structure of their (type) arguments. They are closely related to the concept of parametric polymorphism [5] which abstracts the nature of the processed elements uniformly. Even in bounded polymorphism, like Java Generics [2], there is not much customization that can be done in the behaviour of the instantiated code, depending on the actual type or compile-time values used as parameters.

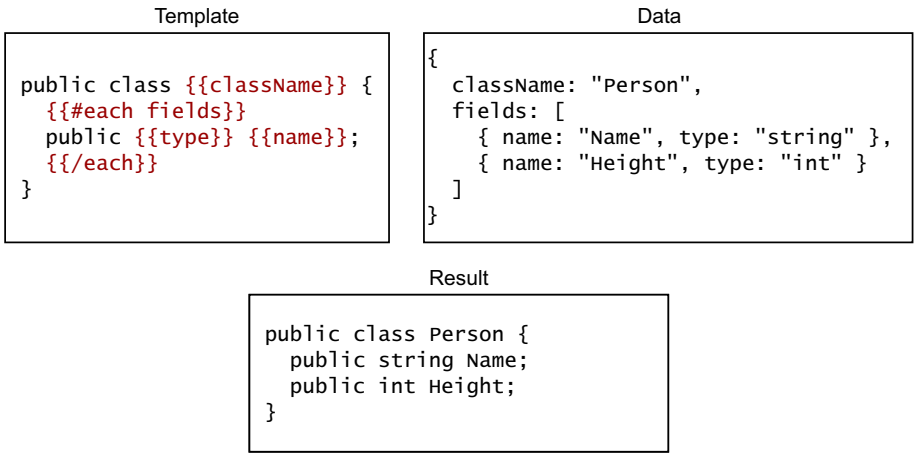
Compile-time computations represented in mainstream languages provide a greater expressiveness in adapting rich pre-prepared code templates to the context of an application. Both MetaOCAML [15], and Template Haskell [32] are generative approaches supporting the algorithmic construction of programs at compile-time. Richer type-level computations have been proposed in generalized algebraic data types [7], and more recently in [3], using refinements kinds, with the capability of reasoning about the structure of types and producing custom code constructions. Our approach is inspired by the latter. The innovation of this work is the use of type-level computation in low-code models, accompanied by the verification of the template code, and the (partial) guarantee that all well-typed instantiations produce well-typed code. The formal proof of completeness of this process is ongoing work and is out of the scope of this presentation. We start by having operations that reason about the structure of types and other model elements and aim to verify more sophisticated conditions on types used as arguments in future work.

UML Templates

Standard UML templates introduce the common concepts of abstraction and parametrization [17] in UML models [25], with some authors extending it and instantiating it in EMF-based tools [6,34,35]. The semantics of these models includes the substitution of parameters and cloning model elements to produce other diagrams. Our approach includes a full-fledged template language with iteration and conditional annotations and a strongly typed approach that can soundly adapt to the environment and provides safety properties to the instantiation. We also provide a formal semantics for a staged template expression language for property setting expressions that is directly implemented in the prototype. Similar verification results can be obtained using approaches like OCL [12], like in [35], or using contracts [8].

In both the approaches [8,35], it is not clear how to verify, at compile-time, both the instantiation of model elements and the expressions being produced for the model instance. By having a conservative extension of the base model, where all parameters must have default values, our base model also supports the partial instantiation of parameters that is present in UML templates. Unlike UML templates, our templates are also ground models that can be viewed, edited, and compiled by production versions of the platform, including its IDE. This also accounts for a graceful evolution of the existing tool ecosystem. Moreover, we evaluate our approach in real-world application of model templates, like [34], but targeting OutSystems platform instead. We add rich compile-time computations to new elements and use their properties.

Fig. 16 Handlebars template



Our approach is a two-stage language [10], that is in a way similar to MetaDepth [11], extending the DSL of the OutSystems platform with an extra "generic" layer. Our aim is to extend single root templates to multiple root instances so that high-level concepts can be instantiated by several parts at a lower-level in the chain of models. The universal language constructs we propose to show/hide model elements, or iterate over compile-time lists of compile-time values (e.g. labels, types, child nodes) are present in MetaDepth code generators that instantiate the lower-level. A crucial difference between the two approaches, and other works with UML models [12,34], is that the verification of model conformance is usually performed on the instances after the substitution of the parameters instead of statically verifying the template and the application of the arguments at compile time. Another difference is that our model uses the same template mechanisms to produce actual runtime values for model element properties.

Template Languages for UI (Web)

Textual template languages have long been used to simplify the development of web applications [28]. They enable the creation of dynamic pages by multidisciplinary teams consisting of web designers (who focus on the static aspects of the web pages) and developers (who are concerned with defining the dynamic elements that fetch and process data).

Many of these template languages, such as ASP.NET and Java Server Pages, allow intermixing imperative code with the template content, while others such as Handlebars [13] and Mustache [21] take a simpler and cleaner approach where the templates are purely declarative. OSTRICH draws inspiration from the latter. In Fig. 16, we present an example of a Handlebars template for generating a Java class, and the result of running it against a concrete data input. Handlebars tags (highlighted in red) are interspersed among text that is copied verbatim to the output. Handlebars attaches no meaning to the verbatim text, and thus can be used to produce anything that can be represented in a text format. However, it is up to the template developer to guarantee that the template will produce well-formed results, which is not a trivial task since the template itself is usually not well-formed with respect to the target language grammar and, consequently, the target language development tools cannot be used to edit and validate the template. OSTRICH addresses these concerns by design by guaranteeing that only well-formed models are produced. The fact that templates are annotated model elements makes it possible to evolve existing tools to support defining templates.

8 Future work

We identify a set of extension points for this work which include some solutions for the limitations identified in Sect. 6.1.

Our template mechanism is at this point partially integrated into the IDE, which means that the ground components are expanded into the current application context, and can be expanded and tweaked to meet the specific needs of the developer. This process disables the possibility of experimenting and changing different templates in the same context, and hinders the effect that evolutions of template definitions may have in their instances. We plan to tackle the template instantiation process in the near future, thus allowing to instantiate template inside template definitions and also to customize the resulting instances, without loosing the hability of reapplying the template or changing its arguments without losing the customizations.

9 Conclusions

In this paper, we present the second iteration of a rich template language for the OutSystems platform, called OSTRICH. The original version of OSTRICH [18] conservatively extends the existing OutSystems model with template

annotations to denote parameters, node properties, special iteration and conditional behaviour to model nodes and template expressions. OSTRICH is a two-stage language that evaluates template nodes and the corresponding annotations to ground model nodes with concrete property values that correspond to a standard OutSystems model. Ground model nodes contain runtime expressions that ultimately define the runtime behaviour of an application.

In our second iteration, we increased the expressiveness of OSTRICH by accounting for dependencies between template parameters. The impact of this extension is twofold. First, more sample code fragments can be captured now. For instance, by enabling dependencies between entities it's now possible to define templates that express master-detail patterns. Second, there is an improved user experience as the instantiation result is more precise which, in turn, reduces the number of changes that need to be done by the application developer.

We also equip our template language and the corresponding template expression language with a verification mechanism that guarantees that all staged expressions are well-formed and will produce results of the right type.

We evaluated our approach by porting existing screen templates available in the OutSystems platform and produced parametrized versions of the same professionally produced components to be used in a safer, easier and faster way.

Acknowledgements We thank the anonymous reviewers for their comments that helped improve the paper. This work is partially supported by FCT/MCTES grants UIDB/04516/2020, PTDC/CCI-INF/32081/2017, and GOLEM Lisboa-01-0247-Feder-045917.

References

1. Atlas Group. Atlas transformation language. https://wiki.eclipse.org/ATL/User_Guide
2. Bracha, G.: Generics in the Java programming language (2004). https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf
3. Caires, L., Toninho, B.: Refinement kinds: type-safe programming with practical type-level computation. In: Proceedings of the ACM on Programming Languages, 3(OOPSLA), October 2019. UID/CEC/04516/2019 PTDC/EEICTP/4293/2014
4. Cardelli, L.: Phase distinctions in type theory (1988). https://www.microsoft.com/en-us/research/publication/phase-distinctions-in-type-theory/
5. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–523 (1985)
6. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G.: An OCL formulation of UML2 template binding. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (Eds) UML 2004—The Unified Modeling Language. Modeling Languages and Applications, pp. 27–40. Springer, Berlin (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, MODELS '09, pp. 644–649. Springer, Berlin (2009)
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 (2003)
10. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001)
11. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Model. **12**(3), 453–474 (2013)
12. 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) (Special issue on Experimental Software and Toolkits)
13. Handlebars-minimal templating on steroids (2021). https://handlebarsjs.com/. Last visited in 11 May 2021
14. Jetbrains. Jetbrains meta programming system (2020). http://github.com/JetBrains/MPS
15. Kiselyov, O.: The design and implementation of BER MetaOCaml-system description. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming—12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4–6, 2014. Proceedings, vol. 8475 of Lecture Notes in Computer Science, pp. 86–102. Springer (2014)
16. Kitchenham, B.A., Mendes, E.: Software productivity measurement using multiple size measures. IEEE Trans. Softw. Eng. **30**(12), 1023–1035 (2004)
17. Liskov, B., Guttag, J.: Abstraction and Specification in Program Development. MIT Press, Cambridge (1986)
18. 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)
19. Lourenço, H., Eugénio, R.: TrueChange™ under the hood: how we check the consistency of large models (almost) instantly. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 362–369 (2019)
20. Lourenço, H., Tavares, J., Eugénio, R., Lourenço, M., Simões, T.: LUV is not the answer: continuous delivery of a model driven development platform. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (2020)
21. Mustache-logic-less templates (2021). https://mustache.github.io/. Last visited in 11 May 2021
22. McConnell, S.: Code Complete—A Practical Handbook of Software Construction, 2nd edn. Microsoft Press (2004)
23. Mendix. Page templates (2021). https://docs.mendix.com/refguide/page-templates
24. Mohagheghi, P., Conradi, R.: Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empir. Softw. Eng. **12**(5), 471–516 (2007)
25. Modeling language specification version 2.5.1 (2017). https://www.omg.org/spec/UML. Last visited in 09 May 2021
26. OutSystems. OutSystems screen templates (2020). https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen_Templates
27. OutSystems. Platform overview (2021). https://www.outsystems.com/platform/
28. Parr, TJ.: Enforcing strict model-view separation in template engines. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds) 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)

29. Pech, V., Shatalin, A., Voelter, M.: JetBrains MPS as a tool for extending Java. In: Plümicke, M., Binder, W. (eds) 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)
30. Perez De Rosso, S., Jackson, D., Archie, M., Lao, C., McNamara, B.A. III.: Declarative assembly of web applications from predefined concepts. In: Masuhara, H., Petricek, T. (eds) Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23–24, 2019, pp. 79–93. ACM (2019)
31. Seco, J.C., Ferreira, P., Lourenço, H.: Capability-based localization of distributed and heterogeneous queries. J. Funct. Program. **27**, e26 (2017)
32. Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: Proceedings of the 2002 Haskell Workshop, Pittsburgh, pp. 1–16 (2002)
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Eclipse Series, 2nd edn. Addison-Wesley, Upper Saddle River (2009)
34. Vanwormhoudt, G., llon, M., Caron, O., Carré, B.: Template based model engineering in UML. In: Syriani, E., Sahraoui, H.A., de Lara, J., Abrahão, S. (eds) 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)
35. 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)

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.

Carla Ferreira received a Ph.D. 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 proje ct on programming tools for decentralised 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.

João Costa Seco graduated in 1993, got a Masters in 1997 and got his Ph.D. 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 centred on the use of programming language-based approaches for automated programming and software evolution to better enable software development processes, adva nce 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 M.Sc. degree from the NOVA School of Science and Technology—NOVA University Lisbon (2022). During her master's, she formalised 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.