OSTRICH - A Type-Safe Template Language for Low-Code Development

Conference Paper · October 2021

DOI: 10.1109/MODELS50736.2021.00030

3 authors, including:

Hugo Lourenço
OutSystems
18 PUBLICATIONS **106** CITATIONS
SEE PROFILE

Carla Ferreira
Universidade NOVA de Lisboa
88 PUBLICATIONS **1,377** CITATIONS
SEE PROFILE

OSTRICH - A Type-safe Template Language for Low-code Development

Hugo Lourenço
OutSystems
hugo.lourenco@outsystems.com

Carla Ferreira
NOVA LINCS
NOVA University of Lisbon
carla.ferreira@fct.unl.pt

João Costa Seco
NOVA LINCS
NOVA University of Lisbon
joao.seco@fct.unl.pt

***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 (re)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.

***Index Terms*—metamodel templating, typechecking templates, low-code, development productivity, model reuse**

I. 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 [17], [18], [24] 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 in a single place all the aspects of their applications, 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 [14], [21]. Code reuse is a well-known practice in software construction in general and essential in any healthy development process [16], [19]. The core factor is abstraction, leading to parametrization, isolation, and information hiding. To this end, OutSystems provides 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 lack of success in replicating and adapting sophisticated code is usually a strong detractor.

In this paper, we address the lack of abstraction and parametrization mechanisms in the OutSystems model-driven approach. The terminology "templates", in low-code platforms and web development in general, does not fully correspond to the technical concept of parameterized code templates but the cloning and modification of samples. The use of such sample "Screen templates" is a common practice in the OutSystems platform [23] and other low-code platforms [20]. An example of a sample screen template is a page 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 on 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 by 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 ap-

plication and a zero-cost abstraction (no overhead at runtime).

We adopt a model-driven approach and conservatively extend the OutSystems metamodel which allows for a seamless integration in the existing IDE, both for 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 its 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 [3], thus avoiding dependencies between compile-time and runtime expressions.

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. 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 (section II), and present its metamodel (section III).
- A semantics for the model, via an instantiation algorithm (section IV), and for the enclosed expression language with embedded compile-time computations (section V).
- An experimental evaluation, obtained by porting sample screens used in cloning and rudimentary adaptation mechanisms native in the OutSystems platform (section VI).
- A critical review of template languages and models for programming languages and models (section VII).
- A set of clear extension points and supporting features for more abstraction, adaptation, verification, and evolution mechanisms (section VIII).

Lastly, we close the paper with some final remarks.

II. 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 section III.

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.

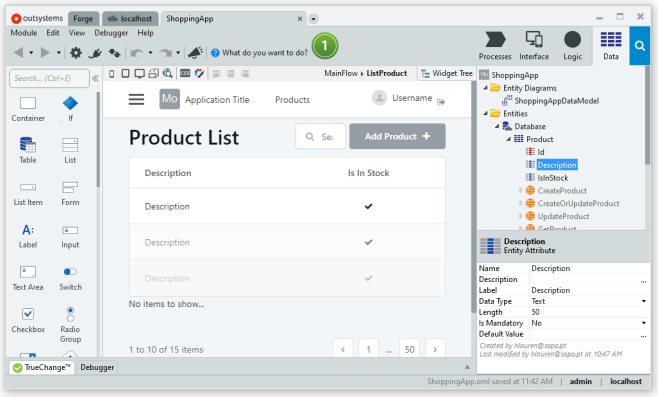

Fig. 1: Product List application.

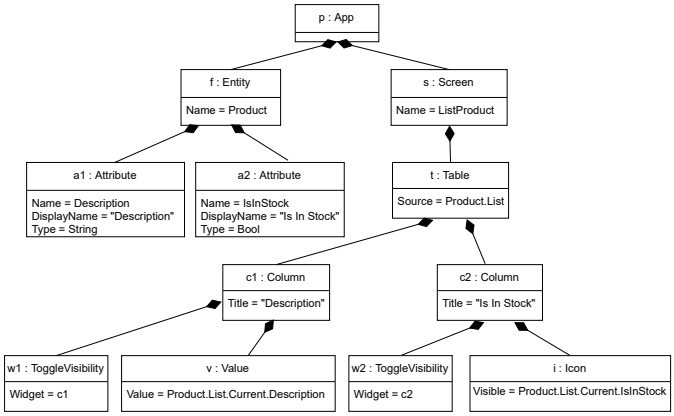

Fig. 2: Product List application model.

In Figure 1 we depict an OutSystems application in our IDE. In Figure 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 parameterized by the 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 Figure 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` (Figure 4).

Under our approach, a template consists of a parameterized annotated model that, when removing the highlighted elements

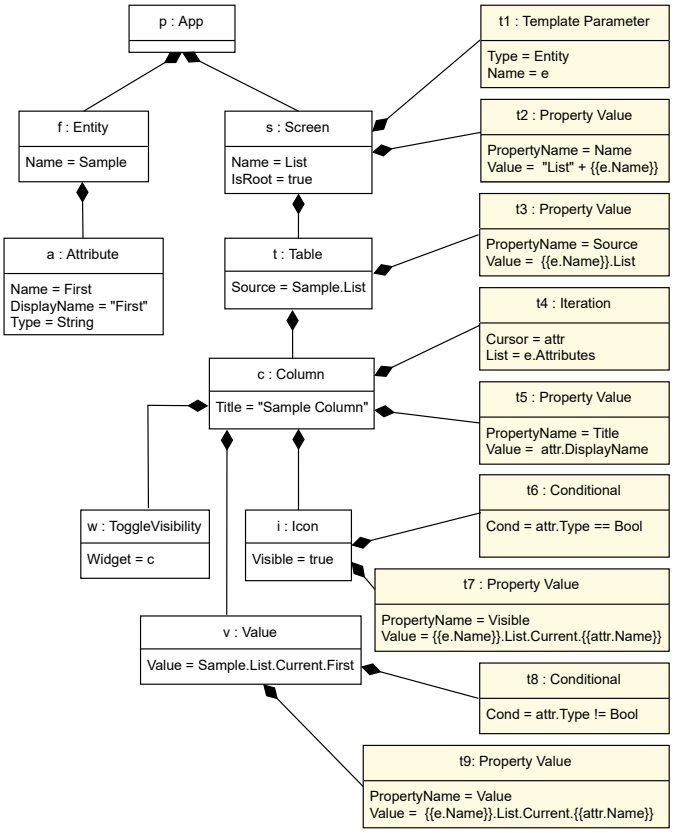

Fig. 3: List template model.

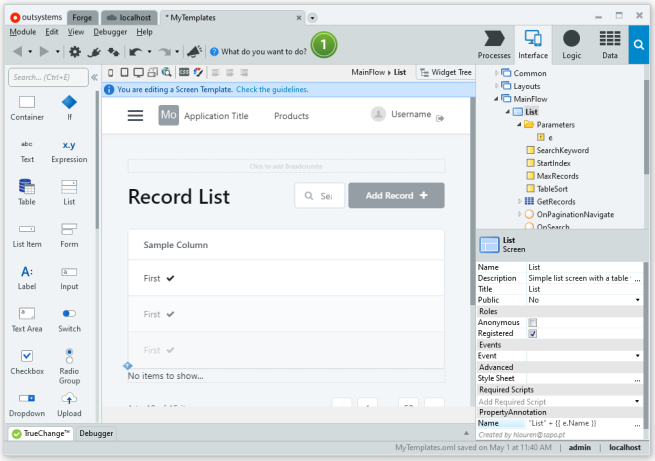

Fig. 4: List template.

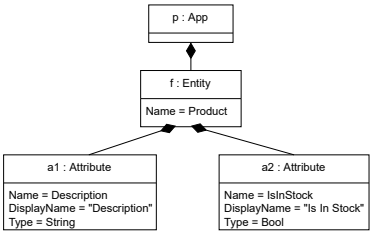

Fig. 5: Target application.

in Figure 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 by an actual entity provided by the developer. For that purpose the template defines a parameter named `e` of type Entity. 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.
- *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 / 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 Figure 2 is obtained by instantiating the template from Figure 3 in the target app of Figure 5, using App `p` as the target location and entity `Product` as the template's argument.

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 does 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 will 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 [18]. 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 datatypes 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.

III. Template Metamodel

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

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 to edit template annotations. In Figure 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). Obvious disadvantages include that the creation of these templates require the costly extension of the existing tools.

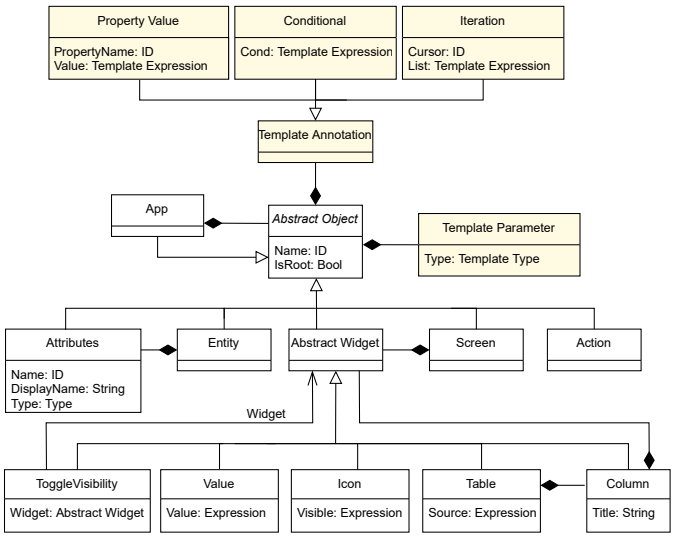

Fig. 6: Template metamodel.

IV. 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 values range from basic value literals to model objects. Figure 2 depicts the result of instantiating the template screen depicted in Figure 3 with the following inputs:

- template: Screen `s` (Figure 3)
- targetParent: App `p` (Figure 5)
- arguments: $\{$ `t1` $\mapsto$ `f` $\}$, with the template parameter `t1` assigned to entity `f` of the caller context

Algorithm 1 starts in line 2, by creating the environment `env` containing all the arguments of the instantiation (using `args`) and the empty map `newObjs` to store the objects created during the instantiation process. 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. Function Evaluate, used to evaluate template expressions, is discussed in detail in section V.

The Traverse function, in line 5, evaluates all Conditional and Iteration annotations present at each template node. For the case of Conditional annotations this amounts to evaluating the annotation's condition and skipping the node if the condition evaluates to `false`. In the case of iteration annotations it first evaluates and iterates the annotation's list. Each iteration is evaluated in a newly created environment that binds the cursor name to the element's list being iterated. This makes the value available for any template expression contained in

Algorithm 1 Template instantiation algorithm

```
    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, newObjs ← newEnv(args), {}
 3:     TRAVERSE(template, parent, env, newObjs, createObject)
 4:     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
 5: function TRAVERSE(templNode, targetParent, env, newObjs, fun)
 6:     if hasConditionalAnnotation(templNode) then
 7:         if evaluate(getCondExpression(templNode), env) == true then
 8:             fun(templNode, targetParent, env, newObjs)
 9:         else skip
10:     else if hasIterationAnnotation(templNode) then
11:         list ← evaluate(getListExpression(templNode), env)
12:         cursorName ← getCursor(templNode)
13:         for all item in list do
14:             env ← beginScope(env)
15:             bind(env, cursorName, item)
16:             fun(templNode, targetParent, env, newObjs)
17:             env ← endScope(env)
18:     else fun(templNode, targetParent, env, newObjs)
19: function CREATEOBJECT(templNode, targetParent, env, newObjs)
20:     obj ← createChild(targetParent, typeof templNode)
21:     newObjs ← newObjs ∪ {(templNode, getCursorsState(env)) ↦ obj}
22:     for all child in getChildren(templNode) do
23:         TRAVERSE(child, obj, env, newObjs, createObject)
24: function SETPROPERTIES(templNode, targetParent, env, newObjs)
25:     obj ← get(newObjs, (templNode, getCursorsState(env)))
26:     for all prop in getProperties(templNode) do
27:         value ← evaluateProperty(templNode, prop, env, newObjs)
28:         setPropertyValue(obj, prop, value)
29:     for all child in getChildren(templNode) do
30:         TRAVERSE(child, obj, env, newObjs, setProperties)

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

the template node and its children. Notice that this process is parametric in function `fun`, that is called for all template nodes, the only difference being the environment used, which is extended in the case of iterations.

The CREATEOBJECT function, 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. E.g., the Iteration annotation `t4` in Figure 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 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`. I.e., after running the first step of the instantiation algorithm we have that:

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

Finally, function SETPROPERTIES is used in the second traversal to set the property values for the all newly created objects. This function starts by looking up the target object in `newObjs` and iterates and sets each of the object's properties. 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 default value. As a special case, if the value is a model object then we try to use `newObjs` to map it. For example, consider ToggleVisibility widget `w` in Figure 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 (Figure 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 compile-time computation of literal values, but also the construction of runtime expressions that will use runtime values of the existing model elements.

V. TEMPLATE EXPRESSIONS

The example of section II illustrates the definition and application of templates in our model. The semantics of the model includes the interpretation of node annotations, along with the compile-time checking and evaluation of the expressions that can be used in such annotations.

The template expression language is a multi-stage language. Its semantics defines a compile-time stage of evaluation whose results are runtime expressions, which is a strict subset of the template language. The language draws the boundaries between type-level and compile-time computations and runtime computations, preservation of the phase distinction [3] results from the introduced typing discipline. Each template expression evaluates in the first stage using an environment containing references to elements in the actual model, parameters introduced in the template declarations, and cursor values introduced by Iteration annotations. Runtime expressions include references to elements like entities, attributes, and aggregates which are usually the anchors to refer to the

state of the program. The second stage evaluation is performed in a closed setting, where all referred elements are inhabited by model elements, and all the expressions associated to element properties are resolved to runtime values.

Before presenting the syntax and semantics of template and runtime expressions, we will start by looking at an example. Consider the template expression used in the property box of List template (cf. Figure 4), and present in the template model (Property Value annotation `t2` shown in Figure 3),

$$\texttt{"List"} \ \texttt{+} \ \{\{\texttt{e . Name}\}\} \tag{1}$$

where `e` is the template's parameter and is of type Entity. In this particular case, expression (1) is used to customize the screen name based on the name of the actual 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) obtained by the parser of Figure 7 is as follows:

$$\texttt{"List"} \ \texttt{+} \ (\texttt{NameOf}\ e) \tag{2}$$

The concrete syntax is designed to be an expression language with embedded template language snippets. The "dot" operator inside the handlebars is context dependent, and is here translated to the native operator (`NameOf .`). We show later that outside of the handlebars, the selection operation is translated to a runtime selection ($\hat{.}$).

Given an instantiation for the free variable e in expression (2) with entity **Product**, that has for the name property the identifier **Product**, we obtain the expression

$$\texttt{"List"} \ \texttt{+ Product} \tag{3}$$

that results in the identifier **ListProduct**. The evaluation of expression (3) is possible because the concatenation operator (**+**) is able to join a String literal with an identifier to define a new identifier. We show later in subsection V-A the detailed evaluation steps for this template expression.

We now present the syntax of the language, defined by the grammars in Figure 7. The language encompasses a sub-language of value literals with lists and records, which are only included in the form of literals and language destructors. 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 selection of 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 [8], [26]. The second stage of evaluation defines the domain of runtime expressions given by the syntax of Figure 7.

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

$$\texttt{attr . DisplayName} \tag{4}$$

v	::=		(**value literal**)
	|	num	(number literal)
	|	$string$	(text literal)
	|	$bool$	(boolean literal)
	|	$[\overline{v}]$	(list)
	|	$\{\overline{L = v}\}$	(record)
V	::=		(**model element**)
	|	$Entity\langle N, T\rangle$	(entity element)
	|	$Attribute\langle N, L, T, \{\overline{L' = v}\}\rangle$	(attribute element)
M	::=		(**term**)
	|	v	(value literal)
	|	x	(variable)
	|	L	(labels)
	|	ID	(name)
	|	$typeid$	(type)
	|	V	(model element)
	|	$M \oplus M$	(value operation)
	|	$M \hat{\oplus} M$	(expression)
	|	$\texttt{NameOf}\ M$	(name property)
	|	$\texttt{LabelOf}\ M$	(label property)
	|	$\texttt{TypeOf}\ M$	(type property)
E	::=		(**expression**)
	|	v	(value literal)
	|	ID	(name)
	|	x	(variable)
	|	$E \oplus E$	(value operation)
	|	$E\ \texttt{.}\ L$	(field selection)

Fig. 7: Syntax of template multi-stage language.

in annotation `t5` of Figure 3. For brevity, we only present entities and attributes. An entity is represented by its name, and a record type defining the name and type of each attribute. An attribute is defined by the name of the entity it belongs to, a label, a type, and a set of properties.

The first stage of template expressions evaluation is expressed by function $[\![M]\!]_{Env}$, defined in Figure 8, and function $(\!|M|\!)$, omitted from the presentation, and that represents the standard interpretation of operations on literal 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 Figure 9. Notice 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 Figure 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.

A. An example of evaluating a template expression

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

$$\{\{\texttt{e . Name}\}\}\texttt{ . List . Current . }\{\{\texttt{attr . Name}\}\} \tag{5}$$

$$[\![v]\!]_{Env} = v \quad \text{(E-VAL)}$$
$$[\![x]\!]_{Env} = Env(x) \quad \text{(E-VAR)}$$
$$[\![L]\!]_{Env} = L \quad \text{(E-LABEL)}$$
$$[\![ID]\!]_{Env} = ID \quad \text{(E-NAME)}$$
$$[\![T]\!]_{Env} = T \quad \text{(E-TYPE)}$$
$$[\![M \,.\, M']\!]_{Env} = (\!|\, (\texttt{PropsOf}\ V) \,.\, L \,|\!) \quad \text{(E-FIELD)}$$
with $[\![M]\!]_{Env} = V$ and $[\![M']\!]_{Env} = L$
$$[\![M \oplus M']\!]_{Env} = (\!|v \oplus v'|\!) \quad \text{(E-VALOP)}$$
with $[\![M]\!]_{Env} = v$ and $[\![M']\!]_{Env} = v'$ and $\oplus \neq .$
$$[\![M \mathbin{\hat{.}} M']\!]_{Env} = [\![M]\!]_{Env} \,.\, L \text{ with } [\![M']\!]_{Env} = L \quad \text{(E-SEL)}$$
$$[\![M \mathbin{\hat{\oplus}} M']\!]_{Env} = [\![M]\!]_{Env} \oplus [\![M']\!]_{Env} \text{ with } \hat{\oplus} \neq \mathbin{\hat{.}} \quad \text{(E-EXP)}$$
$$[\![\texttt{NameOf}\ M]\!]_{Env} = \texttt{NameOf}\ [\![M]\!]_{Env} \quad \text{(E-NAMEOF)}$$
$$[\![\texttt{LabelOf}\ M]\!]_{Env} = \texttt{LabelOf}\ [\![M]\!]_{Env} \quad \text{(E-LABELOF)}$$
$$[\![\texttt{TypeOf}\ M]\!]_{Env} = \texttt{TypeOf}\ [\![M]\!]_{Env} \quad \text{(E-TYPEOF)}$$

Fig. 8: Semantics of template expressions.

$$\texttt{NameOf}\ \mathit{Entity}\langle N, T\rangle \triangleq N$$
$$\texttt{LabelOf}\ \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle \triangleq L$$
$$\texttt{TypeOf}\ \mathit{Entity}\langle N, T\rangle \triangleq T$$
$$\texttt{TypeOf}\ \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle \triangleq T$$
$$\texttt{PropsOf}\ \mathit{Attribute}\langle N, L, T, \{\overline{L' = v}\}\rangle \triangleq \{\overline{L' = v}\}$$

Fig. 9: 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 show in Figure 7 is the following

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

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

$$[e = \mathit{Entity}(\textbf{Product}, \{\textbf{Description} : \mathit{string}, \textbf{IsInStock} : \mathit{bool}\}),\ \mathit{attr} = \mathit{Attribute}(\textbf{Product}, \textbf{IsInStock}, \mathit{bool}, \{\textbf{DisplayName} = \texttt{"Is In Stock"}\})\,]$$

where the 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 Figure 8 as follows,

$$[\![(((\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List}) \mathbin{\hat{.}} \mathit{Current}) \mathbin{\hat{.}} (\texttt{LabelOf}\ \mathit{attr})]\!]_{Env} =$$
$$[\![((\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List}) \mathbin{\hat{.}} \mathit{Current}]\!]_{Env} \,.\, [\![\texttt{LabelOf}\ \mathit{attr}]\!]_{Env} = \quad \text{(E-SEL)}$$
$$[\![((\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List}) \mathbin{\hat{.}} \mathit{Current}]\!]_{Env} \,.\, \texttt{LabelOf}\ [\![\mathit{attr}]\!]_{Env} = \quad \text{(E-LABELOF)}$$
$$[\![((\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List}) \mathbin{\hat{.}} \mathit{Current}]\!]_{Env} \,.\, \textbf{IsInStock} = \quad \text{(E-VAR)}$$
with $Env(attr) = \mathit{Attribute}(\textbf{Product}, \textbf{IsInStock}, \ldots)$
$$[\![((\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List})]\!]_{Env} \,.\, [\![\mathit{Current}]\!]_{Env} \,.\, \textbf{IsInStock} = \quad \text{(E-SEL)}$$
$$[\![(\texttt{NameOf}\ e) \mathbin{\hat{.}} \mathit{List}]\!]_{Env} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \quad \text{(E-LABEL)}$$
$$[\![\texttt{NameOf}\ e]\!]_{Env} \,.\, [\![\mathit{List}]\!]_{Env} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \quad \text{(E-SEL)}$$
$$[\![\texttt{NameOf}\ e]\!]_{Env} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \quad \text{(E-LABEL)}$$
$$\texttt{NameOf}\ [\![e]\!]_{Env} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} = \quad \text{(E-NAMEOF)}$$
$$\textbf{Product} \,.\, \mathit{List} \,.\, \mathit{Current} \,.\, \textbf{IsInStock} \quad \text{(E-VAR)}$$
with $Env(e) = \mathit{Entity}(\textbf{Product}, \ldots)$

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

Consider now the evaluation of template expression (2):

$$[\![\texttt{"List"} + (\texttt{NameOf}\ e)]\!]_{Env} =$$
$$(\!|\, [\![\texttt{"List"}]\!]_{Env} + [\![(\texttt{NameOf}\ e)]\!]_{Env} \,|\!) = \quad \text{(E-VALOP)}$$
$$(\!|\, \texttt{"List"} + [\![(\texttt{NameOf}\ e)]\!]_{Env} \,|\!) = \quad \text{(E-VAL)}$$
$$(\!|\, \texttt{"List"} + (\texttt{NameOf}\ [\![e]\!]_{Env}) \,|\!) = \quad \text{(E-NAMEOF)}$$
$$(\!|\, \texttt{"List"} + \textbf{Product} \,|\!) = \quad \text{(E-VAR)}$$
with $Env(e) = \mathit{Entity}(\textbf{Product}, \ldots)$
$$(\!|\, \textbf{ListProduct} \,|\!) = \textbf{ListProduct} \quad \text{(def. of + and } (\!|\,|\!) \text{ function)}$$

Finally, the evaluation of expression (4) is:

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

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.

B. Typechecking

A good development experience is highly dependent on the soundness of the templating system. Ensuring soundness of the model and the underlying expression language basically means that any instantiation of a template, which is a compile-time computation, must produce a valid runtime model. This concern is not in the core of all the templating languages, especially the ones targeting user interfaces, because the target languages are very flexible and untyped. This is not the case when composing and generating full-fledged expressions. In a low-code setting it is not admissible to have to tweak the instantiated result to be able to execute without errors.

We define a strongly-typed template expression language, whose type system ensures that all template expressions always produce valid runtime expressions. A valid template expression always produces values that match the expected datatype for the corresponding properties and annotations in the model elements. This feature is extensive to the full

metamodel guaranteeing that any sound instantiation of a template results in a sound program.

Our typed template expression language captures the correct use and correct typing of model element's names, and labels of element properties and record typed expressions. The template expression language is a staged language [8], [26]. This means that there is a clear separation, driven by the type system, between the instantiation-time construction of expressions, and the execution of those expressions with runtime values. We use singleton values to create dependencies between labels and entities, and ensuring that all uses of labels are part of the target entity or record type. The type system also ensures the convergence of template expressions, which means that instantiation of templates always terminates.

VI. 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 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 template's structure. One of the design goals for `OSTRICH` is to support the conversion of such screen templates into proper parameterized 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. Figure 10). To evaluate our work, we selected the top ten most instantiated screen templates (Table I) 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 pre-defined *sample* entities that the developers must replace by 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 or used/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 II summarizes our results on the number of changes made when adapting each one of the screen templates. Notice the uniformity on 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 datatypes. 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 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.

A. Limitations

While converting the screen templates, we identified a few limitations in the use of `OSTRICH` in more advanced scenarios. It is not yet possible to specify richer constraints on the parameter values other than typechecking or to express dependencies (or relations) between parameters. E.g., for Table and Form templates a common requirement is to select not only the entity, but also the subset of attributes to be displayed. Although `OSTRICH` allows one to define a *list of attributes* as a parameter, there is currently no way to require them to be associated with one particular entity or template parameter.

Template instantiation is still a *unidirectional* process. It works by expanding the template definition and executing the annotations in the target context. After instantiation, it is not possible to re-apply a template by changing the arguments or to account for the evolution of the template definition. Developers must start from scratch and will lose any manual customization. We are currently addressing these issues.

VII. Related Work

Model to Model transformations: Templating adds an abstraction mechanism to the OutSystems language, cf. functions in general purpose languages. Thus, templates are edited at the same abstraction level as regular OutSystems code, using the same development tools, thus dramatically increasing development productivity. Model to Model transformations (M2M) in EMF [28] or MPS [12] require a skilled "language designer" to encode them to text or a lower-level model.

We argue that our approach cannot be directly encoded in any known M2M tool. We do define a kind of M2M

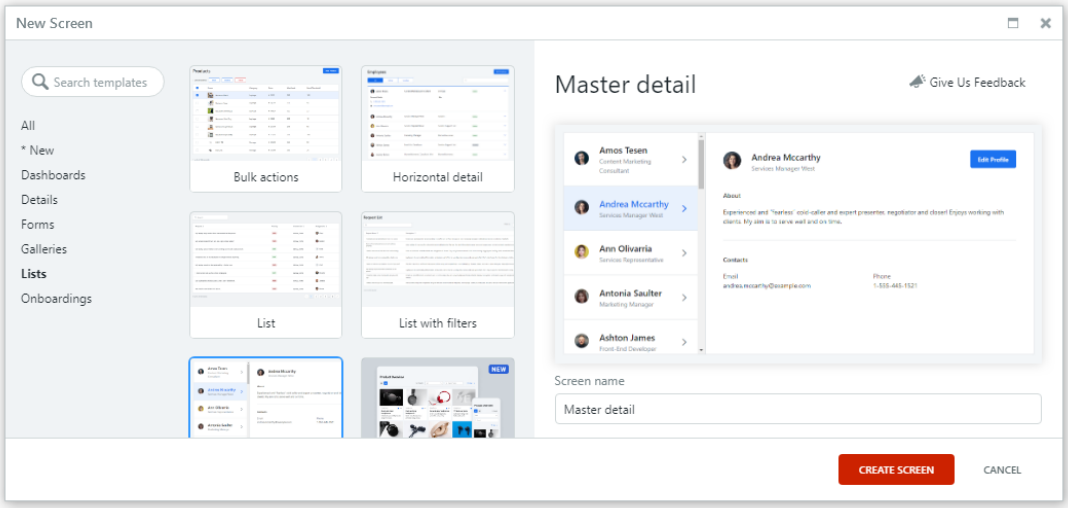

Fig. 10: New Screen dialog, showing the available Screen Templates.

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

TABLE I: Top ten most used "screen templates".

		Annotations		
Screen template	Parameters	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

TABLE II: Screen templates adaptation statistics.

transformation, but the constraints proposed, and our prototype checks go well beyond syntactic soundness and usual integrity constraints in OCL. Our model aims to verify template code and produce well-typed programs in low-code.

General purpose programming languages: Standard template languages take little advantage of the structure of their (type) arguments. They represent the concept of parametric polymorphism [4] which abstracts the nature of the elements being processed in unformly. Even in bounded polymorphism, like Java Generics [1], there is not much customization that can be done in the behaviour of the instantiated code.

Compile-time computations represented in mainstream languages provide a greater expressiveness in adapting rich pre-prepared code templates to the context of application. Both MetaOCAML [13], and Template Haskell [27] are generative approaches supporting the algorithmic construction of programs at compile-time. Richer type-level computations have been proposed in generalized algebraic data types [6], and more recently in [2], 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 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 [16] in UML models [22], with some authors extending it and instantiating it in EMF-based tools [5], [29], [30]. 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 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 [10], like in [30], or using contracts [7].

In both the approaches [7], [30], it is not clear how to verify,

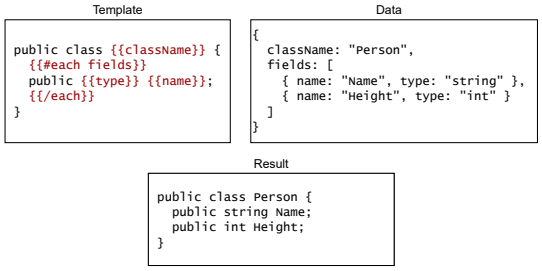

Fig. 11: Handlebars template

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 accounts also for a graceful evolution of the existing tool ecosystem. Moreover, we evaluate our approach in real-world application of model templates, like [29], but targeting OutSystems platform instead. We add rich compile-time computations to new elements and use their properties.

Our approach is a two-stage language, that is in a way similar to METADEPTH [15], 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 [10], [29], 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 [25]. 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 to intermix imperative code with the template content, while others such as Handlebars [11] and Mustache [9] take a simpler and cleaner approach where the templates are purely declarative. OSTRICH draws inspiration from the latter. In Figure 11 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 produced 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.

VIII. CONCLUSIONS

In this paper we present a first approach to a rich template language for the OutSystems platform, called OSTRICH. This model conservatively extends the existing model with template annotations, to denote parameters, node properties, special iteration and conditional behaviour to model nodes, and template expressions. Our language 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.

We 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 parameterized versions of the same professionally produced components to be used in a safer, easier and faster way.

As future work we have identified a set extension points and solutions for the limitations identified in subsection VI-A. Our template mechanism is at this point 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 track the template instantiation process, allow to customize the instances, and still be able to reapply the template without losing the customization.

Furthermore, the restrictions on the arguments can be strengthened to capture dependencies between parameters. For instance, we plan to properly detect and restrict patterns like relations between two entity-typed parameters.

ACKNOWLEDGEMENTS

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

References

[1] Gilad Bracha. Generics in the java programming language, 2004.

[2] Luis Caires and Bernardo Toninho. Refinement kinds: type-safe programming with practical type-level computation. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019. UID/CEC/04516/2019 PTDC/EEICTP/4293/2014.

[3] Luca Cardelli. Phase distinctions in type theory. January 1988.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[5] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. An OCL formulation of UML2 template binding. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 — The Unified Modeling Language. Modeling Languages and Applications*, pages 27–40, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[7] Arnaud Cuccuru, Ansgar Radermacher, Sébastien Gérard, and François Terrier. 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, page 644–649, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001.

[9] Mustache - logic-less templates, 2021. https://mustache.github.io/. Last visited in 2021-05-11.

[10] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1):27–34, 2007. Special issue on Experimental Software and Toolkits.

[11] Handlebars - minimal templating on steroids, 2021. https://handlebarsjs.com/. Last visited in 2021-05-11.

[12] Jetbrains, 2020.

[13] Oleg Kiselyov. The design and implementation of BER MetaOCaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing.

[14] Barbara A. Kitchenham and Emilia Mendes. Software productivity measurement using multiple size measures. *IEEE Trans. Software Eng.*, 30(12):1023–1035, 2004.

[15] Juan Lara and Esther Guerra. From types to type requirements: Genericity for model-driven engineering. *Softw. Syst. Model.*, 12(3):453–474, July 2013.

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

[17] Hugo Lourenço and Rui Eugénio. TrueChange™ under the hood: How we check the consistency of large models (almost) instantly. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 362–369, 2019.

[18] Hugo Lourenço, Joana Tavares, Rui Eugénio, Miguel Lourenço, and Tiago Simões. LUV is not the answer: continuous delivery of a model driven development platform. *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020.

[19] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004.

[20] Mendix. Page templates. https://docs.mendix.com/refguide/page-templates, Mar 2021.

[21] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Softw. Engg.*, 12(5):471–516, October 2007.

[22] Modeling language specification version 2.5.1, 2017. https://www.omg.org/spec/UML. Last visited in 2021-05-09.

[23] OutSystems. OutSystems screen templates. https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen_Templates, Oct 2020.

[24] OutSystems. Platform overview. https://www.outsystems.com/platform/, 2021.

[25] Terence Parr. Enforcing strict model-view separation in template engines. In Stuart Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 224–233. ACM, 2004.

[26] João Costa Seco, Paulo Ferreira, and Hugo Lourenço. Capability-based localization of distributed and heterogeneous queries. *Journal of Functional Programming*, 27:e26, 2017.

[27] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, pages 1–16, October 2002.

[28] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2 edition, 2009.

[29] Gilles Vanwormhoudt, Mathieu Allon, Olivier Caron, and Bernard Carré. Template based model engineering in UML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 47–56, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Gilles Vanwormhoudt, Olivier Caron, and Bernard Carré. Aspectual templates in UML. *Software & Systems Modeling*, 16(2):469–497, 2017.