A Model Driven Approach for Unifying User Interfaces Development

Henoc Soude
Institut de Mathématiques et des Sciences Physiques, BENIN

Kefil Koussonda
Epitech, FRANCE

***Abstract*—In this paper, we dealt with the rapid development of client web applications (frontend) in a context where development frameworks are legion. In effect with the digital transformation due to the COVID-19 pandemic we are witnessing an ever-increasing demand of the application development in a relatively short time. To this is added the lack of skilled developers on constantly evolving technologies. We therefore offer a low-code platform for the automatic generation of client web applications, regardless of the platform or framework chosen. First, we defined an interface design methodology based on a portal. We then implemented our model driven architecture which consisted of defining a modeling and templating language, centered on user data, flexible enough to not only be used in various fields but also be easily used by a citizen developer.**

***Keywords*—Model driven; user interface; modeling language; templating language; low code; citizen developer**

I. Introduction

Over the past two decades, web applications have evolved and are now used in all areas of everyday life: medical, transport, e-commerce and social networks. One of the reasons for their success is that they can be used on all our devices like laptops, smart phones, tablets and desktops. The interfaces are determining elements in the acceptance of a web application [1], [2]; their development is therefore of paramount importance hence the multiplicity of frameworks in the market. For some years the frameworks *javascript* have spread widely to the point of becoming indispensable in the development of modern applications. By the way Sacha Greif et al. [3] have launched an annual survey since 2016 to analyze developments and trends in *javascript* frameworks by focusing on based data from more than 20,000 developers spread over more than 100 country. This study shows us, among other things, the most used frameworks (react, angular, vue) but doesn't give us any information on how they choose, knowing that they have different learning curves more or less complex. Indeed, the COVID-19 pandemic requires us to rapidly develop more and more complex applications, in the context of a global shortage of developers [4], [5].

Model-based development (MDD) has been used in the literature [6]–[8] in order to simplify the development of applications and increase productivity. It is a development paradigm which according to Stephen et al. [9], consists in building the model of a system and then generate a real instance of this system from the obtained model. MDD was quickly adopted by the Object Management Group (OMG) which standardized model-based architecture (MDA) and is composed of several levels, two of which are essential in our development context of web application: platform independent model (PIM); used to describe the system and platform specific model (PSM); used to generate an instance actual system. The PIM and PSM models are described using languages that are either standardized or domain-specific (DSL). Most of these languages are neither suitable, nor flexible enough and too complex [10], in terms of learning curve.

One of the solutions to address the problems related to developer shortage is the use of low-code platforms [11], which in addition to using model-based approaches, use graphical or high-level languages to make development accessible to all. According to a Gartner report more than 50% of companies will adopt them as a solution strategic by 2023 [12]. There are many low code [13]–[15] which agrees with generating interfaces from models. They have the particularity of either generating code for their own platform or to use languages specific to their domain wich most of the time are difficult to handle. Added to this is the fact that the adoption of low-code platforms by citizen developers is conditioned by the simplicity and accessibility of the underlying models.

This work is part of a more global project to implement a low code [16] platform to automate the development of tasks in various fields. For the sake of consistency, all projects share not just the same development languages: *c/c++* and exchange data: *json*, but also the same automation system; the generator must generate codes in several domains without requiring the intervention of expert developers.

The general research question of our work is: *How can we unify user interface (UI) development?*. Whatever framework used: Mithril, React, Vue, the development process must be identical and the learning curve should be as simple as possible even for a person without any developing skills (citizen developer). The solution we propose is to automatically generate the code for the different platforms from a single model, defined by the user. It comes in the form of the following contributions:

- the development of a portal and the definition of an UI design methodology based on a portal. It allows to gain in productivity and to make the implementation of UI simpler since their navigation elements have to be configured.
- the development of a cloud-based graphic editor for UI modeling
- the definiton of a modeling language which has simple structure; all its elements have the same basic,

and flexible structure; it can be extended for new platforms.

- the development of a code generation system consisting of an engine and a template language.

The rest of the document is organized as follows: Section II presents state of the art. Then we described our model-based approach to Section III. Our interface modeling language and our template language are described in Sections IV and V. In Section VI we presented a general discussion of work then we concluded with the Section VII

II. Related Work

Whatever platform or model-based approach used, they are distinguished by the modeling language and the generation system used.

A. UI Modeling Language

The principle of modeling a user interface consists in finding an abstract representation of the constituent elements of the interface. This representation usually contains information relating to the organization, the description of the elements in the interface and interactions with users. This explains the multiplicity of modeling languages that differ depending on the type and storage of the information they contain.

UIML [17] is the first universal modeling language independent of the platforms and technologies used. It is an XML dialect which describes an interface in five sections: the description, the structure, the data, the style, and the event. Subsequently the language was standardized by OASIS group. USIXML [18] is another language based on XML and standardized by the World Wide Web Consortium (W3C) whose particularity is to describe the multimodal interfaces. The proposed language in our work differs from the two languages presented by the format of the definition of models; we use the *json* format unlike the XML used in most templates [19], [20].

Brambilla *et al.* [21] present interaction flow modeling language (IFML) which was quickly adopted as a standard by OMG. Language allows to describe the structure and interactions with end users via a set of visual components representing the different elements of an interface. This concept of visual modeling has greatly appealed whether in the academic world or that of industry [22]–[26]. The language has introduced a high level of abstraction that makes an element of the model we can match several components of an interface. This ambiguity is resolved by the language when generating the interface, since the user provides these correspondence elements. This constitutes a real problem in the works as shown where we have no idea of the interface that the end user wants to generate: we cannot provide a mapping that will satisfy all users.

More recently Moldovan *et al.* [27] presented a model-based approach for developing user interfaces f' or multi-target applications. They first define their modeling language (OpenUIDL) whose syntax is based on the JSON format. Then they present the different stages of their approach for the development of interfaces: the design, the generation and the deployment of the interface. Although dealing with the same problem with an almost identical approach, our work differs mainly in the level of the semantics of the modeling language and the process of generation. In order to improve the problem of limited accessiblity to defintions of modeling languages (cf. [27]), we propose a model of a simpler and more intuitive level than the authors of [27]. Indeed in their model a node can be of type static value, dynamic reference, element, conditional, repeat, slot, nested-styled and its content is specified through the attribute of the same name. So you need at least two objects *json* nested to define an element of an interface then in our model the equivalent of the node directly represents an element of the interface. Their code generation model remains limited to web interfaces and cannot allow "citizen developers" to generate code or data in specific areas contrary to what we propose.

B. Code Generation

In the context of automatic web interface generation, two approaches are often used: that based on the structures tree abstracts [27] and that of the engines template. The major drawback of the first approach is that it requires having technical skills if you want to generate code in a language other than the original one. We are indeed in need of a system that allows seasoned users or not to generate code or documents in various fields.

The second approach is more appropriate in our context since the engines offer a template description language; the user will only have to describe his new model through this language. Most languages used for the development of java web interfaces (freemaker [28]), python (jinja2 [29]) and javascript(mustache [30]) all have template engines whose languages does not tell us nor inhibit us from defining the models of views whose structure is not known during design. XSLT [31] is a language standardized by W3C which allows you to transform XML documents into another format (HTML, XML, js,etc.). It easily solves the problem mentioned above thanks to these directives*template* and *apply*; unfortunately there remains a verbose language and very complex to handle.

III. Model based Approach

The objective of our work is to allow developers to implement web applications without worrying about platforms or frameworks available in the market. For this we propose, as indicated in the Fig. 1 a three-step approach: design, build, and refinement of the interface after deployment. Except for the refinement, the other two steps are MDD classics. The distinctiveness of our solution lies in the methodology, the concepts, the languages and tools we use in the different stages.

A. UI Design

The first principle of our design approach is that the user doesn't waste time implementing common and recurring concepts in web applications: menu navigation and operation application on user data. For this we have implemented an application portal which not only is an application receptacle but also has mechanisms for specifying menu and application-related actions. As shown, Figure 2, is divided into five parts: (i) the main bar which indicates the name of the current application and different menus drop-down such as the list

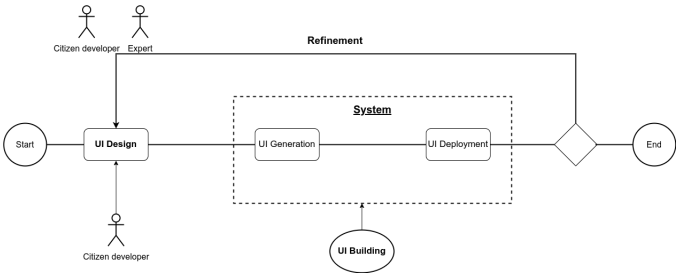

Fig. 1. The Different Steps of our Model-based Approach.

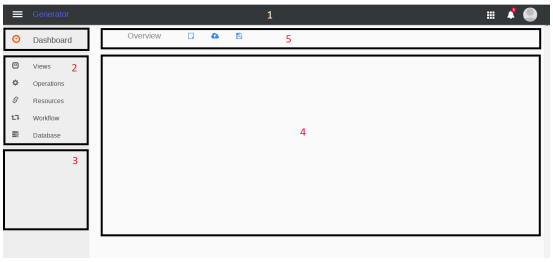

Fig. 2. The Structure of our Application Portal.

of applications, (ii) represents the elements of menu of the current application, (iii) represents the list of applications related to the current application, (iv) represents the workspace in which the different views of the application will be hosted and (v) represents the action bar which indicates the possible operations on a given view. The design of an application's interfaces therefore consists of identifying and declaring the set of views or interfaces that compose it and for each of it to define its content and its operations.

The second principle of our approach is that of the separation of concerns that we have implemented in our editor interface graphic: users of different level techniques will be able to intervene on different parts as shown in Fig. 3 the design is divided into six sections: *view*, *template*, *variables*, *functions*, *actions* and *lifecycle*.

The *view* section describes the organization of a view in the application portal. The form elements `inside menu`, `role` and `icon` allow resp. to specify whether the view is accessible or not in the application menu, the role needed to access the view and view icon in the menu. The `modal` attribute indicates that the view can be imported by another view.

The *template* section describes the contents of a view. The user uses the different predefined elements to describe the content of its view. The `tag` can be used to specify non-predefined elements. Every Adding of an element, the user can choose either to see the preview (tab interface) or to see the generated model (tree tab). At any moment of the design the user can select an element of the model and modify its properties.

The *variables* section describes the variables used by the different view elements. They are usually created by the user which can also associate test values to then. The *functions* section describes user-defined methods. The *actions* section describes sight operations. The user must provide the icon as well as the method to call. Finally the *lifecycle* section

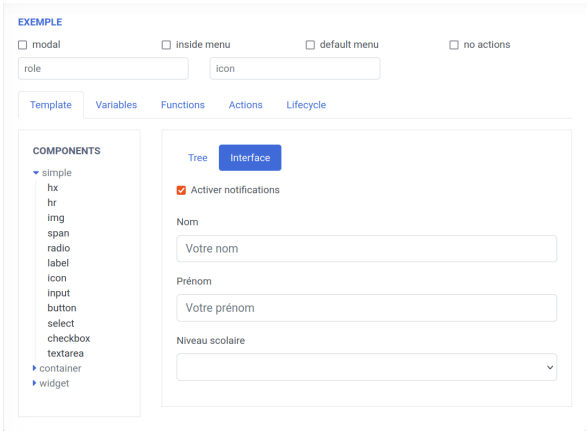

Fig. 3. The Interface Confuguration of a View.

describes the mechanisms of *javascript* component status notification.

B. UI Building

The user does not intervene in the process of generating the views even if he is the instigator. As shown in Fig. 4, our template takes as parameter the view model and the mapping of the elements of the seen. The mapping of an element corresponds to its html template for a support given. We have therefore provided the mapping of all the predefined elements of our editor for the following frameworks: mithril.js, react.js, vue.js. An example of mapping is presented in Section V-B.

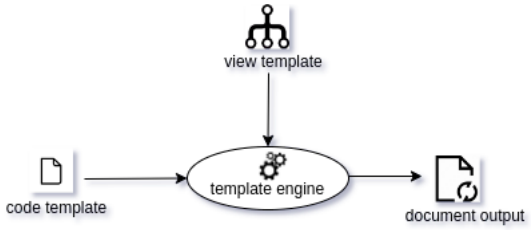

Fig. 4. Our Code Generation Process.

In addition to specificities related to the generation of web applications our build system needs to be flexible enough to not only allow generation in multiple domains but also allow citizen developers to edit their own model for specific needs. For this we have proposed a model language based on text and simple substitution directives, intuitive and non-verbose. We have also decided to limit the number of instructions to make it easier to get started. We have implemented a simple and adaptable algorithm to all situations. The algorithm consists of traversing a tree structure (in *json* format) and each node of the tree generates the corresponding code by retrieving the associated template to its type. This implies that any *json* object passed as a parameter must have the `type` attribute.

C. UI Refinement

After the generation of the interfaces, the user can deploy the application on our beta server to be tested by all connected

users. This mechanism allows us to integrate the Agile principles in our view generation process: the end users or team members can make feedback that will be integrated during this phase.

The positive point of our refinement mechanism associated with principle of the separation of concerns developed during the phase of design is that it is possible for a citizen developer to design the organizational aspect of the views: positioning of the different interface elements. Once his work has been validated then a user with technical skills will be able to finalize the views during the refinement phase.

IV. UI Modeling Language

The most natural representation of an interface is that of a tree in which a node corresponds to an element of the interface. The Fig. 5 is a representation in the form of a tree of the html component `table`: it has `tr` elements which, in turn, have `td` elements. Our modeling therefore consists in defining in a format *json* the structure of interfaces. The nodes of the description tree are represented by the objects*json* whose structures are defined via their json schema.

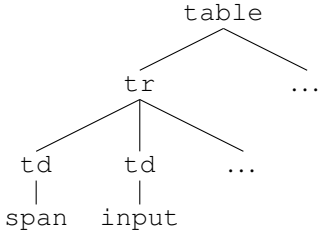

Fig. 5. Representation of `table` Component as a Tree.

A. Syntax of Models

Fig. 6 presents the schema *json* of the basic structure of elements of an interface. An element is described by its type and by either its value or its descendants. All other properties of the element are optional; our generator is able to generate values for them by default. The `type` attribute is a character string whose value is the name of the element with which the object is associated. The `children` attribute is an array that describes the children of an element. The attribute `value` is a character string containing the value of an element. The semantics of the value of an element depending on its type. For example the value of a `button` is the text that accompanies it while the value of a `span` represents its content.

The value of an element can either be literal or come from a variable. For this reason we use a formatting which consists of preceding the value of the attribute `value` by one of the following characters: `$, #, %` . When the attribute value is not prefixed for any of these characters then the value is used as such. When preceded by the character# then it should be treated as given *json*. When she is preceded by the character `$` then the value of the element comes from a variable, defined inside the html component, whose name is the value of the attribute without the prefix character. When the attribute value is preceded by the character `%` then the value of the element comes from a variable passed as a parameter to the html component. In the following example {``a": ``foobar", ``b": ``$foo", ``c": ``%bar", ``d": ``#{}"}, `a, b, c, d` have the respective value `foobar`, the value of the variable`foo`, the variable of the variable `bar` passed as component parameter and an empty json object.

```
{
  "$id": "schema/base",
  "type": "object",
  "properties": {
    "type":    {"type": "string"},
    "value":   {"type": "string"},
    "children":{"type": "array",
    "items":   {"$ref": "#"}}
  },
   "required": ["type",
     {"oneOf": ["value", "children"]}]
 }

```

Fig. 6. The Basic Structure of an Element of our Model.

```
{
    "$id": "schema/view",
    "type": "object",
    "properties": {
        "type": {"type": "string"},
        "imports":  {"type": "array",
            "items": {"$ref": "#/$defs/keyval"}},
        "variables":{"type": "array",
            "items": {"$ref": "#/$defs/keyval"}},
         "functions":{"type": "array",
             "items": {"$ref": "#/$defs/keyval"}},
         "template": {"type": "object",
             "properties": {"$ref": "schema/base"}},
         "oninit":   {"type": "string"},
         "oncreate": {"type": "string"},
         "onupdate": {"type": "string"},
         "onremove": {"type": "string"}
     },
     "required": ["type", "variables",
                "modals", "functions", "template"],
     "$defs": {"keyval":{
         "type": "object",
         "properties": {
           "name"  {"type":"string"},
           "value" {"type":"string"}
         }
     }}
 }

```

Fig. 7. Metamodel of the View Component.

```
{
    "$id": "schema/input",
    "type": "object",
    "properties": {
      "type":       {"type": "string"},
      "value": {"type": "string"},
      "subtype": {"enum":["text", "password",
                  "date", "email", "number", "file"]}
      "label":   {"type": "string"},
       "placeholder": {"type": "string"},
       "class":   {"type": "string"},
       "style": {"type": "string"},
       "onchange": {"type": "string"}
     },
     "required": ["type", "subtype", "value"],
     "additionalProperties": true
 }

```

Fig. 8. Metamodel of Input Component.

```
{
    "$id": "schema/tag",
    "type": "object",
    "properties": {
      "type":  {"type": "string"},
      "value": {"type": "string"},
      "class": {"type": "string"},
      "style": {"type": "string"},
      "children": {"type": "array",
          "items": {"$ref": "schema/base"}},
    },
    "required": ["type", "value", "children"]
}
```

Fig. 9. Metamodel of the Tag Component.

```
{
    "$id": "schema/list",
    "type": "object",
    "properties": {
      "type": {"type": "string"},
      "data": {"type": "string"},
      "root": {"type": "string"},
      "class":{"type": "string"},
      "style":{"type": "string"},
       "iterator": {"type": "string"},
       "children": {"type": "array",
          "items": {"$ref": "schema/base"}},
    },
    "required": ["type", "data", "children"]
}
```

Fig. 10. Metamodel of List Component.

```
{
    "$id": "schema/wizard",
    "type": "object",
    "properties": {
      "type": {"type": "string"},
      "done": {"type": "string"},
      "data": {
        "type": "array",
        "items": {"$ref": "#/$defs/pagedef"}},
    },
    "required": ["data"],
    "$defs": {"pagedef":{
        "type": "object",
        "properties": {
          "name" {"type":"string"},
          "content" {"type":"string"}
        }
    }}
}
```

Fig. 11. Metamodel of Wizard Component.

B. Components

We have predefined a number of interface elements which we have organized into three categories:

- *simple*; they have no child elements. `hr`, `hx`, `img`, `span`,`radio`, `label`, `icon`, `input`, `button`, `select`, `checkbox`, `textarea`.
- *container*; they have child elements. `nav`, `tab`, `link`, `list`, `form`, `group`, `table`, `accordion`, `dropdown`, `paragraph`.
- *widgets*; these are non-standard html elements; `wizard`, `treeview`, `carousel`, `display`.

We have not implemented all the elements of an interface but rather those that come up regularly in web applications. Nevertheless the `tag` element can be used to implement elements not predefined.

In the rest of the document we present only the special components (`view` and `tag`) and one component per category due to the simplicity of the model.

a) view: An element of type `view` is a special element allowing to model the organization of interfaces in the application portal (see section III-A). As shown in its json schema, in Figure 7, the attributes `variables`, `imports`, `functions` and `template` are mandatory. The `template` attribute is an object representing the content of sight. The attributes `variables`, `imports` and `functions` represent respectively the variables, the imported elements and the user-declared functions.

b) input: Fig. 8 represents the model definition of a `input`. The attributes `value` and `subtype` are required. This model represents multiple types of `input` via the `subtype` attribute; other attributes are added to the model depending on the type chosen. The label and the placeholder can also be specified via attributes of the same name. The user can specify the action to perform when changing the value via the `onchange` attribute.

c) tag: The `tag` type element is used to model dynamic attribute (unknown at design time) or non-predefined elements. As shown in Fig. 9 the attributes `value` and `children` are mandatory. The `children` attribute contains the definition of the element's children given while the attribute `value` represents the real no of the element. The element name can be assigned literally or via a variable (cf. Section IV-A).

d) list: The `list` type element is used to model components from a repetitive action; this is the case for example of the elements `dl`, `ul` and `table`. As shown in Fig. 10 the attributes `data` and `children` are required. `children` contains the definition of the elements that are repeated according to the data coming from the `data` attribute which must be an array. The `root` attribute when present becomes the parent element containing the repeated elements. The `iterator` attribute represents the name of the iterator to use in the code for traversing data in `data`.

e) wizard: Fig. 11 represents the model definition of a `wizard`. The attribute `data` is mandatory and represents all the pages of the wizard. Pages are defined by an object whose attribute `name` represents the page name and the `content` attribute represents the content of the page. The content of a page is a reference that is resolved when generated. The action to be performed after the process is completed is specified via the `done` attribute.

V. Template Language

The language that we propose must not only make it possible to write the model codes of our web interfaces but also be sufficiently simple and flexible to allow a citizen developer to write their own models in various areas. For this we have chosen a language composed of texts: they are copied as such by the template engine, and substitution directives: they are replaced by the value of the variables or expressions they represent.

In order to make the language simple and accessible, the transformation of directives is solely based on user data; unlike some languages [28], [31], ours does not allow for defining variables or functions in our models.

User data is passed as a parameter to the template engine as an object of *json* (see Fig. 12) whose attributes are the variables used by the substitution directives. The language uses the symbols `{,},[,]` as block delimiters, the character `%` to escape special characters and the character `$` to introduce directives.

```
{
  "string": "foobar", "number": 12.23,
  "bool": false, "null": null,
  "object": {"key": "val"},
  "array":  [10,20,30]
}
```

Fig. 12. An Example of User Data.

A. Directives

a) content: This directive is used to retrieve the value of an attribute of the data passed as a parameter to the engine.

$$\$ < attribut >< . < attribut_1 >< \cdots . < attribut_n >>> \quad (1)$$

$$\$ < attribut > [< index >] \quad (2)$$

Its syntax is presented by the expression 1 where $< \ attribute \ >$ represents the attribute whose value we are looking for. Considering the data of Fig. 12, the directives $\$string$, $\$number$, $\$bool$, $\$ull$, $\$object$ and $\$array$ respectively have the value `foobar`, `12.23`, `false`, `null`, an empty string and an empty string. The directives $\$object$ and $\$array$ have the value of empty strings since we must specify the element that is sought within them. With regard to the directive $\$object$ we use the optional part of the expression 1 by preceding the name of the element with the character `.`. The $\$object.key$ directive is used to retrieve the value of the `key`. The expression notation 2 is used for arrays. The pattern $< \ index \ >$ indicates the position of the element of the array that we are looking for; in our example $\$array[2]$ and $\$array[\$number]$ have the value `20` and an empty string since $\$number$ is not an integer.

The evaluator of a directive returns the Boolean value `false` which indicates that an error occurred while evaluating and the value `true` in the opposite case. In case of error the evaluator generates an empty string.

b) alternative: This directive allows you to choose between two models. The expression 3 presents its syntax where $||$ represents the separator of the two patterns. The model $< \ model1 \ >$ is generated when it tests true while model $< \ model2 \ >$ is generated when it tests true when the first shows false. In considering the Figure data 12 the guidelines $\$string||\$number$ and $\$foo||nothing$ have the value `foobar` and `nothing`.

$$< model1 > || < model2 > \quad (3)$$

c) test: This directive makes it possible to choose between two models according to the value of a variable. The expressions 4 and 5 present its syntax.

$$\$ < attribut > \{< model1 >\}\{< model2 >\} \quad (4)$$

$$\$ < attribut >< CMP >< VAL > \{< model1 >\}\{< model2 >\} \quad (5)$$

In the expression 4 the test is implicit; the model $< \ model1 \ >$ is generated if attribute `attribute` is true else $< \ model2 \ >$ is generated. The attribute is considered true when it exists and value is neither `false` nor `null`. In the expression 5 we specify the comparator as well as the reference value. The comparator can be one of the operators following: $=, <, >, >=, <=, ! =$. Considering the data of Fig. 12 the directives $\$foo\{found\}\{not \ \ found\}$ and $\$number = 12.23\{equal\}\{not \ \ equal\}$ have value `not found` and `equal`.

d) repeat: This directive makes it possible to repeat the generation of the model according to the number elements in an array variable. The expression 6 presents its syntax where $< \ attribute \ >$ must be of type `array`.

$$\$ < attribut > *\{< model1 >\} \quad (6)$$

$$\$\$ < . < attribut >> \quad (7)$$

The expression 7 presents the iteration variable notation that represents an array element. When the element is an object then we use the optional part of the expression in order to specify the attribute to be to find the value. Considering the data in Figure 12 the directive $\$array * \{\$\$\}$ has the value `102030`.

e) separator: This directive is used to automatically generate separators in the context of the *repeat* directive. The expression 8 presents its notation where $< SEPATOR >$ is the separator character. Considering the data of the Fig. 12 the directive $\$array * \{\$\$\$:\}$ has the value `10:20:30`.

$$\$ < SEPATOR > \quad (8)$$

f) only if: This directive is used to generate a model if and only if all its directives evaluate to true. The expression 9 presents its notation. Considering the data in Fig. 12 the directives $[\$string \ has \ \$number]$ and $[\$name \ has \ \$foo]$ have for value `foobar has 12.23` and an empty string.

$$[< model >] \quad (9)$$

g) built-ins: These are language variables whose notation is present by the expression 10 where $< \ func \ >$ represents the name of the variable and $< arg_i >$ represent the arguments associated with the variable. Arguments are optional and allow to change the behavior of the variable.

$$\$ < func >< , < arg_1 >, \cdot, < arg_n >> \$ \quad (10)$$

The variable `$index$` is used in a context of the directive *repeat* and returns the iteration number. The directive $\$array*$

{$*index*$} has the value `012` if we consider the data of the Fig. 12.

The variable `$call$` is used in a context of the directive *repeat* and allows recursively calling the template engine on the elements of a table. Elements must be objects that have at least one attribute *type*.

The variable `$href$` is used when we want to retrieve data on media different from the object passed as a parameter to the engine. The data user must contain an *urn* attribute that describes the exact location of the data. The *urn* is composed of fragment separated by the character `:`. The first fragment can take the value `file`, `db`, `rest` depending on whether the data comes from a file, a database or a rest resource. The variable `$count$` is used in a context of the directive *repeat*; it returns the number of elements in an array. The directive $\$array * \{\$count\$\}$ has the value `3` if we consider the data of the Fig. 12.

The variable `$sum$` is used in a context of the directive *repeat*; it returns the sum of the elements in an array. The directive $\$array * \{\$sum\$\}$ has the value `60` if we consider the data of the Fig. 12.

h) Arithmetic operations: The language allows performing arithmetic operations only on the user data according to the syntax of the expression 11 where $< OP >$ Represents an operation among `+,-,/,*,%` and $< VAL >$ represents a whole value. Operations are only allowed for attributes of type *number* Or *string*. Only the addition operation is possible on strings; that is to reduce, from the beginning, the size of the string by the number $< VAL >$. The directives $\$string + 3$ and $name + 3$ have the value `bar` and `name+2`.

$$\$ < attribut >< OP >< VAL > \tag{11}$$

```
module.export= ()=>{
  $modals*{var $$.name%=$href$;}
  $variables*{var $$.name%=$$.value;}
  var $name%_cls = function(){
  return {
    $functions*{$$.name:$$.code$,}
  };}();
  return {
    oninit:(vnode)=>{
      app.setActions($noaction{null}{
        %[$actions*{$call$$,}%]
      });
      $oninit
    },
    [oncreate:(vnode)=>%{$oncreate%},]
    [onupdate:(vnode)=>%{$onupdate%},]
    view:(vnode)=>{
     return $template*{$call$};
    }
    [,onremove: (vnode)=>{$onremove}]
  };
}
```

Fig. 13. Mapping of `view` component for Mithril Framework.

```
<template>
  $template*{$call$};
</template>
<script>
$modals*{import {$$.name} from "$vref+5";}
export default {
  data() {
    return {
      $variables*{$$.name: $$.value$,}
    }
  },
  computed: {
    $functions*{$$.name$$.code$,}
  },
  beforeCreate(){
    app.setActions($noaction{null}{
      %[$actions*{$call$$,}%]
    });
    $oninit
  },
  [created(){$oncreate},]
  [updated(){$onupdate},]
  [destroyed(){$onremove}]
};
</script>
```

Fig. 14. Mapping of `view` Component for vue Framework.

B. Mapping

Mapping consists of using the language to describe the structure of the code to be generated for each element of our model. Here we present only the Mapping of the `view` element, as an example. Fig. 13 and 14 present the code structure of `view` For mithril and vue frameworks. Users can provide their mapping during the init phase of the engine.

VI. Discussion

How do we assess the learning curve of our modeling method Views? We compared our modeling approach to the one presented by Moldovan *et al.* [27] and that based on IFML. All the three approaches use graphic elements in order to simplify the Modelization. However, the other two approaches introduce new concepts, in addition to the components of an interface.

Why are we talking about unification when we have not limited ourselves on only three frameworks? We talk about unification because our system allows you to add other platforms or frameworks. Just provide the mapping (cf. section V-B) of the predefined elements for the said platform.

Is our built system as simple as claimed? We submitted, to fifteen students, the writing of templates in different fields using our language and XSLT. After analyzing their return from experience it appears that our work was preferred because of its syntax compact and its low learning curve.

Is our build system flexible enough to be used in different areas? Unlike the work of Moldovan *and Al.* [27] which use one generator per target, we use the same generator ourselves regardless of the target; only the mapping changes. Furthermore our generator was used to generate unit tests in some of our projects.

VII. Conclusion

In this work we presented a model-based approach for the development of web interfaces, whatever the platform and have implemented an application portal to simplify the Modeling work. All elements common to interfaces are implemented in the portal and configured during modeling. Then we implemented a cloud-based graphic editor for the modeling of interfaces. The elements of the interface are described via our model *json* whose flexibility allows implementing the abstraction of elements of different platforms without introducing new concepts. Finally we implemented an automatic code generation system for frameworks Mithril, React, Vue. The system is composed of a template language, with a low learning curve, and a template engine focused on user data: the generator determines the model to generate based on the type of the data. Their association make the generation system suitable for different areas.

To further reduce interface development time, we would like to generate them from their draft.

References

[1] E. Yigitbas, I. Jovanovikj, K. Biermeier, S. Sauer, and G. Engels, "Integrated model-driven development of self-adaptive user interfaces," *Software and Systems Modeling*, vol. 19, no. 5, pp. 1057–1081, 2020.

[2] M. Thomas, I. Mihaela, R. M. Andrianjaka, D. W. Germain, and I. Sorin, "Metamodel based approach to generate user interface mockup from uml class diagram," *Procedia Computer Science*, vol. 184, pp. 779–784, 2021.

[3] (2022) State of javascript. [Online]. Available: https://stateofjs.com

[4] Daxx. (2022) Us and the global tech talent shortage in 2022. [Online]. Available: httaps://www.daxx.com/blog/development-trends/software-developer-shortage-us

[5] T. Sloyan. (2021) Is there a developer shortage? yes, but the problem is more complicated than it looks. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2021/06/08/

[6] S. Abrahão, E. Insfran, A. Sluÿters, and J. Vanderdonckt, "Model-based intelligent user interface adaptation: challenges and future directions," *Software and Systems Modeling*, vol. 20, no. 5, pp. 1335–1349, 2021.

[7] L. Alwakeel and K. Lano, "Model driven development of mobile applications," in *Doctoral Symposium, ECOOP 2020*, 2020.

[8] A. Sabraoui, A. Abouzahra, K. Afdel, and M. Machkour, "Mdd approach for mobile applications based on dsl," in *2019 International Conference of Computer Science and Renewable Energies (ICCSRE)*, 2019, pp. 1–6.

[9] W. B. Frakes and S. Isoda, "Success factors of systematic reuse," *IEEE Software*, vol. 20, no. 05, pp. 14–19, sep 1994.

[10] J. S. Mittapalli and M. P. Arthur, "Survey on template engines in java," in *ITM Web of Conferences*, vol. 37. EDP Sciences, 2021, p. 01007.

[11] A. C. Bock and U. Frank, "Low-code platform," *Business & Information Systems Engineering*, vol. 63, no. 6, pp. 733–740, 2021.

[12] P. Vincent, K. Iijima, M. Driver, J. Wong, and Y. Natis, "Magic quadrant for enterprise low-code application platforms," *Gartner report*, 2020.

[13] (2022) Appian. [Online]. Available: https://appian.com/platform/low-code-development/

[14] (2022) Mendix. [Online]. Available: https://www.mendix.com/

[15] H. Lourenço, C. Ferreira, and J. C. Seco, "Ostrich-a type-safe template language for low-code development," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2021, pp. 216–226.

[16] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178.

[17] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "Uiml: an appliance-independent xml user interface language," *Computer networks*, vol. 31, no. 11-16, pp. 1695–1708, 1999.

[18] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, and M. Florins, "Usixml: a user interface description language for specifying multimodal user interfaces," in *Proceedings of W3C Workshop on Multimodal Interaction WMI*, vol. 2004. sn, 2004.

[19] S. Berti, F. Correani, F. Paterno, and C. Santoro, "The teresa xml language for the description of interactive systems at multiple abstraction levels," in *Proceedings workshop on developing user interfaces with XML: advances on user interface description languages*, 2004, pp. 103–110.

[20] F. Paterno', C. Santoro, and L. D. Spano, "Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, pp. 1–30, 2009.

[21] M. Brambilla and P. Fraternali, *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[22] M. Hamdani, W. H. Butt, M. W. Anwar, and F. Azam, "A systematic literature review on interaction flow modeling language (ifml)," in *Proceedings of the 2018 2nd International Conference on Management Engineering, Software Engineering and Service Sciences*, 2018, pp. 134–138.

[23] A. Huang, M. Pan, T. Zhang, and X. Li, "Static extraction of ifml models for android apps," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2018, pp. 53–54.

[24] N. Yousaf, F. Azam, W. H. Butt, M. W. Anwar, and M. Rashid, "Automated model-based test case generation for web user interfaces (wui) from interaction flow modeling language (ifml) models," *IEEE Access*, vol. 7, pp. 67 331–67 354, 2019.

[25] R. K. Cao and X. Liu, "Ifml-based web application modeling," *Procedia Computer Science*, vol. 166, pp. 129–133, 2020.

[26] N. Kharmoum, S. Ziti, Y. Rhazali, and F. Omary, "An automatic transformation method from the e3value model to ifml model: An mda approach," *Journal of Computer Science*, vol. 15, no. 6, pp. 800–813, 2019.

[27] A. Moldovan, V. Nicula, I. Pasca, M. Popa, J. K. Namburu, A. Oros, and P. Brie, "Openuidl, a user interface description language for runtime omni-channel user interfaces," *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. EICS, pp. 1–52, 2020.

[28] (2015) What is apache freemarker™? [Online]. Available: https://jonas-moennig.de/how-to-cite-a-website-with-bibtex/

[29] (2008) Template designer documentation¶. [Online]. Available: http://jinja.octoprint.org/templates.html

[30] mustache. [Online]. Available: https://mustache.github.io/

[31] xslt cover page. [Online]. Available: https://www.w3.org/TR/xslt/