GenAn – DSL for describing web application GUI based on TextX meta-language

Bojana Zoranovic*, Lazar Nikolic* , Rade Radisic*, Gordana Miosavljevic*, Igor Dejanovic*
* Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
{bojana.zoranovic, lazar.nikolic, radisic.rade, grist, igord}@uns.ac.rs

***Abstract*—Increase in number of internet users in recent years conditioned reportion of work between client and server applications. Client applications had to become richer, aware of data model and states. Front-end developers now have to shape not only views but also states, transitions, communication with server side etc. Each of those requires mastering different formalism that can represent an additional burden for a developer. The main goal of this implementation is to provide a uniform and consistent manner of describing graphical user interface (GUI) for data-oriented web application on higher abstraction level. This paper presents the implementation of domain-specific language along with code generator developed for this language. GenAn DSL is implemented using TextX meta-language. This project aims to enable users to write simple, platform-independent, model-based application descriptions that include GUI specification. Considerable benefit of this approach is that it leads to significant decrease in the number of code lines necessary for application description, higher component reusability and faster development process.**

Keywords: DSL, GUI, code generator, TextX

I. INTRODUCTION

Since the beginning of year 2000th, the number of internet users has grown rapidly [1]. Users felt the necessity for web application rich with interactions and diverse components which required intensifying communication with servers. This trend imposed shifting part of processing, validations and business logic from server-side to client-side applications. The main goal was to diminish the load on server-side which became unable to handle such amount of traffic. Newly acquired complexity brought additional responsibility for developers who need to master different formalisms for describing different client-side application segments. Developers lack a consistent mechanism for application description on high abstraction level.

The goal of this paper was to address following problems:

- Minimizing number of lines of code. This could be solved by specifying model on higher abstraction level.
- Increase code/component reusability. User should be able to specify data model or GUI component once and reuse it later.
- Binding component representation with its type – each data type should have own GUI representation. Implementing this feature would provide necessary flexibility while maintaining language simple.

For this implementation Model-Driven Software Development (MDSD) approach was used. This approach suggests that development foundation should be model instead of code. Domain specific abstractions are used for formal modeling as their use leads to improvement in productivity, quality and ease of maintenance.

Domain specific languages [2] offer increase in productivity and improvement in software quality with high-level abstractions strongly related to the applied domain can partially solve this problem.

Code generator [3] is automated process that accesses models, extracts data and transforms it to code ready to be interpreted or compiled to executable code. Generated code is executable, complete, of production quality and does neither require additions nor adjustments. Meta-model (modeling language with its concepts, semantics and rules) controls the generation. Domain framework [3][4] sets the basis for interpreting software solution and architecture for future development. Among other, domain framework prevents duplication of generated code and enables integration with existing code.

Arpeggio parser [5] is recursive, descending parser with backtracking and memorization (packrat parser). It was developed on the Faculty of Technical Sciences in Novi Sad by a team headed by professor Igor Dejanovic. Arpeggio is available on GitHub under the MIT license. It supports grammar definition using PEG notation or Python programming language. Its primary goal is to provide ground for domain specific language development – more specifically infrastructure for TextX parsing.

TextX [6] is a meta-language for domain specific language development based on Arpeggio parser. As Arpeggio, it was developed on the Faculty of Technical Sciences in Novi Sad. TextX was built to aid textual language development enabling custom language development as well as providing support for already existing language or particular data format.

MEAN [7] is full-stack JavaScript solution for creating fast and robust web applications using MongoDB, Express.js, AngularJS and Node.js.

The source code of GenAn can be obtained at [8]. GenAn is available under the terms of MIT license.

The rest of the paper is structured as follows: in Section II related work has been reviewed. Section III presents

GenAn DSL along with complementing code generator. Section IV contains case study. Section V concludes the paper.

II. RELATED WORK

Recently, the number of commercial solution emerged in the field of DSLs and code generators for client-side application development. Most popular ones allow the user to quickly set the foundation for future work by using existing model.

One of the most widely spread solutions is JHipster [9]. JHipster is Yeoman[10] generator that targets modern web application generation by uniting robust Java stack with Spring Boot on the server-side with one of the popular JavaScript frameworks application on client-side. The accent is on interoperability and support of different database management systems and front-end frameworks. Generated application is optimized, complete and production-ready. Default generation manner is Yeoman style – as a set of console based questions. JDL (JHipster Domain Language[11]) – a DSL for defining the data model is provided to the advanced users. JDL offers decent support for data model description with a precise specification of implementation details. However, it is strongly connected to Java programming language which represents an issue for its wider use. Besides that, JDL does not support custom display on the data level.

Sifu[12] is software tool for fast web and mobile application development developed by DryTools studio from Novi Sad. It is based on Sifu specification language named Sifu DSL which is used for generating AngularJS client side and Java or Scala server side application. Sifu specification language is focused on user interface concepts such as page, form and view, while data model concepts are defined in server side definition.

WebDSL[13][14] is a tool for developing web application with rich data model. Generator for WebDSL was implemented using Stratego/XT[15], SDL[16] and Spoofax[17] language workbench and consists of few smaller sublanguages dedicated for different technical domains. WebDSL is a mature, complex language with great expressive possibilities. Due to its complexity, training time for new users could be a major issue. Some syntax constructions are based on Java language concepts which could be a possible problem as well.

Some implementation analyzed in this section, rely only on the model definition for generating entire client side application. Future adjustments and additions (especially in terms of user interface) must be performed in generated code. Other implementations offer low-level user interface specification but require longer user training time.

III. IMPLEMENTATION

Referent implementation consists of domain-specific language and web application generator. At the time of writing, MEAN stack application generation was supported.

TextX was used as a meta-language which made this language development significantly easier.

A. *GenAn DSL*

Meta-model – grammar determines GenAn DSL abstract syntax. It was developed by combining data model concepts with web application graphical user interface concepts.

The base rule in GenAn DSL grammar is a module. The module consists of concept definition. Basic language concepts are Object, View, Page, Menubar and Sidebar. These concepts could be divided into two major groups:

1. Concepts related to data model description (Object)
2. Concepts related to user interface description (View, Page, Menubar and Sidebar)

```
Module:
    concept += Concept
;
Concept:
    Object|View|Page|Footer|Menubar|Sidebar
;
```

Figure 1. Module and Concept concept definition

Here we will provide a short analysis of basic concepts in GenAn DSL:

Object is a base concept for the data model. Figure 2 shows definition from meta-model. It is defined using keyword object, followed by object name, metadata, properties and queries. Metadata contains data about entity relationships with other entities.

```
Object:
    'object' name=ID ':'
        ('Meta' ':' meta += Meta ';')?
        properties *= Property
        queries *= Query
    ';'
;
```

Figure 2. Object concept definition

Object properties contain data related to the data model (such as name or type) as well as data about GUI representation (label). The user can extend property by inserting additional attributes such as HTML classes for representation, options to choose or hyperlink to photo. Property type can be selected among built-in types – TextX built-in objects [18]. Built-in types are important for both data model and graphical representations.

Concepts dedicated to GUI only are View, Page, Form, Menubar, Sidebar and Footer.

View represent often used segment of user interface that could be composed of other views as sub-views. The main objective for the view concept is to allow designing typical views for future use. An important concept related to the view is selector. The selector is a reference to existing view or form, access to the data model based on properties, data access based on foreign key etc. A part of meta-model used to describe views and sub-view is given in Figure 3.

```
View:
    'view' name=ID ('for' object=[Object]'.'query=[Query])? ':'
        views *= SubView
    ';'
;

SubView:
    RowSeparator | ViewInView
;

ViewInView:
    selector=Selector ('at' position=NUMBER)? ('size' size=NUMBER)?
;
```

Figure 3. View concept definition

Page concept represents autonomous web application page and possesses the name, entity data, title, data about common GUI components, sub-views and their layout. Page definition is given in Figure 4.

```
Page:
    'page' name=ID ('for' object=[Object]'.'query=[Query])?':'
        ('params' '=' urlParams+=UrlParam[","])?
        ('indexPage' '=' indexPage=BOOL)?
        'title' '=' title=STRING
        (using *= Use)
        (layout=Layout)*
        views *= ViewOnPage
    ';'
;
viewinview:
    selector=Selector ('at' position=NUMBER)? ('size' size=NUMBER)?
;
```

Figure 4. Page concept definition

Form is a GUI component dedicated to data manipulation. It is represented as regular HTML form with additions to support required technology. In addition, it contains data about the object for display and set of actions available to user. Form definition is given in Figure 5.

```
Form:
    'form' 'for' obj=[Object] 'actions' '=' actions+=FormAction
;

FormAction:
    'save'|'update'|'remove'|'view'
;
```

Figure 5. Form concept definition

Menubar, sidebar and footer are common GUI components. Menubar and sidebar are displayed as references (hyperlinks). If the menu item leads to generated page, its name should be set with prefix "@" (for example @user_page). The page is referenced by URL by setting attribute ref (for example ref='http://google.com').

The **Menubar** is graphical control element. It contains sub-menus filled with actions that user can trigger or links to pages to which user can shift control. Menubar meta-model is given in Figure 6.

```
Menubar:
    'menubar' name=ID  (brandName=STRING)? (brandLink=ReferencedParameter)? ':'
        ('color' '=' color=Color)?
        menus*=Menu
    ';'
;

Menu:
    ('menu' name=ID (side=Side)? ':'
        items*=RefLink
    ';')| ref = RefLink (side=Side)?
;
```

Figure 6. Menubar concept definition

The **Sidebar** is graphical control element that displays various forms or information to the left or right side of application GUI. In business application, it often shows hyperlinks - shortcuts to frequently used operations. Sidebar definition is given in Figure 7.

```
Sidebar:
    'sidebar' name=ID ':'
        ('color' '=' color=Color)?
        links*=RefLink
    ';'
;
```

Figure 7. Sidebar concept definition

Footer is GUI component position in the bottom of the page, beneath main content and often contains basic data about application such as the name of the company in charge of maintenance and development, current date, license etc. Figure 8. shows footer definition.

```
Footer:
    'footer' name=ID paragraph=STRING (side=Side)? ('color' '=' color=Color)? ';'
;
```

Figure 8. Footer concept definition

B. Code generator

Before code generation, syntax checks are performed. These checks are based on mechanisms supported by TextX known as object processors. Object processors allow concept specific checks for every meta-model concept.

GenAn code generator consists of two independent parts – client and server application generator that communicate over HTTP [19] protocol compliant to REST architectural style [20] which makes them independent and replaceable. The routes are generated automatically according to data entity (object) description from the model. At the moment, only CRUD and filter operations are supported.

Presented software version supports AngularJS [21] client-side and Node.js [22] server-side with MongoDB [23] database (MEAN stack application).

GenAn code generator implements visitor design pattern. From the model, TextX generates Python class tree. Each element carries data about the manner it should be handled and translated to goal language. Visitor design pattern enables uniform tree walk and reduces the load on generator's code by shifting control to particular meta-model classes. The generation process is supported by Jinja2 template engine.

The first step in the generation process is the instantiation of TextX interpreter. In this process, TextX built-in types are instantiated. Fixed built-in types save user's time (the since user does not have to define them in every model) and reduce the number of code lines for model description. In GenAn DSL, built-in types are focused on predefined GUI components – views.

The installation process begins with domain framework instantiation. The framework represents a basis for AngularJS application with configuration files and often used components as a generic module for communication with the server, localization module, GUI layouting module, data transfer module, basic CSS and font files.

Object generation starts with foreign key and relationship name (it should be unique) check. Afterward, a model is transformed by adding a new (meta-)property that represents that relationship along with the view necessary for this relationship display on GUI. Generation process continues with query conversion to the format capable of being sent as HTTP GET method parameter. After that, infrastructure for communication with server-side is provided (AngularJS factory for CRUD and query operation in referent implementation).

Page generation begins with retrieving references to AngularJS factories for objects represented in that page.

After that, page controller is rendered using existing page template. Next step is sub-view layouting for the application GUI by default or specified view position. In this step, page footer, menubar and sidebar are inserted and rendered which may impact other views' positions. The last step in this process is inserting page and sub-view routes in the application routes.

Form generation starts with rendering AngularJS controller component. Every supported operation from form description creates separate controller function. For each object property Bootstrap component enriched by AngularJS validation before being inserted to form view.

Server-side generation process starts with reading configuration files containing mapping data about GenAn types mapping to database types (MongoJS in this case). After that, all necessary node modules are installed. For every object Mongoose schema is created.

For every entity, a separate node route file with own Express router is created which makes every entity a module separate from the rest of the application. Formed route module extends main application routes. Every router's methods is derived from HTTP methods (GET, POST, PUT, DELETE). Each of these methods is mapped to a callback function defined in route_callbacks directory. For each object, a separate file with callback function is formed. Supported functions are:

- **getAll**. Gets multiple instances of demanded object. Supports users' queries from client-side and transforms them to database queries.
- **getById**. Finds demanded instance by the identifier.
- **Post**. Creates an instance from submitted data.
- **deleteById**. Removes the database instance by identifier if it exists.
- **putById**. Changes a database instance by the identifier with submitted instance.

Finally, a portion of the code had to be manually written to adjust generated code to users' demands. The integration method was adapted to AngularJS framework. Different integration technique had to be used for every AngularJS segment. User defines new routes or changes generated ones by creating new routes with existing URL. To redefine a route user needs to set its name, URL, sub-view position, view URL and controller name. An example of redefined route is shown on Figure 9.

```
(function () {
    'use strict';

    angular
        .module('app')
        .constant('newRoutes', {
            'index_new': {
                url: '/index',
                views: {
                    'center': {
                        templateUrl: 'app/views/home/home.html',
                        controller: 'HomeController',
                        controllerAs: 'ctrl'
                    }
                }
            }
        });

})();
```

Figure 9. Redefined route example

Extending AngularJS views with manually written code is fairly simple as it adds up to redefining route with new view.

Redefining AngularJS service and factories is performed by introducing new (user-defined) factories and services instead of generated ones. Generated components stay unchanged, which allows user to tailor existing components to his needs in one part of the application, while using them unchanged in the other.

Integration of manually written and generated code was the most complex for AngularJS controllers. User defines new controller and names it similarly as the generated with prefix "user" (for example home.controller.js and user.home.controller.js). The code generator firstly checks if the manually written controller exists and adds it to the routes. Generated controller is injected into the redefined to expose its methods for reuse and reimplementation. An example is given in Figure 10.

```
(function () {
    'use strict';

    angular.module('app')
        .controller('UserHomeController', UserHomeController);

    UserHomeController.$inject = ['EventBus', '$filter', '$stateParams', '$controller', '$scope'];

    function UserHomeController(EventBus, $filter, $stateParams, $controller, $scope) {
        var ctrl = this;
        //Generisani controller se injektuje u korisnički
        var _super = $controller('HomeController', {$scope: $scope});
        ctrl = angular.extend({}, _super, {"_super": _super});

        ctrl.funkcija = function() {
            //Poziv funkcije iz generisanog controller-a
            ctrl._super.funkcija();

            //Korisnik može da doda sopsteni kod
        };
    }
})();
```

Figure 10. Redefined controller example

IV. CASE STUDY

This section presents web application chosen to showcase solution's possibilities.

Based on GenAn descriptions given in Figure 11. and Figure 10. application was generated.

```
menubar mainMenu 'GenAn' @home:
        menu User:
            'User Form' @user_form_page
        ;
    ;

sidebar mainSidebar:
        'Home' @home
        'GitHub' ref='https://github.com/theshammy/GenAn'
    ;

footer mainFooter 'Copyright MIT, organization GenAn ©' right color=black ;

object user:
    username text label='User name'
    mail email label='Email'
    gender combobox label='Gender' parameters = { 'Male', 'Female' }
;

page user_form_page:
    title='User form'
    use menubar mainMenu
    show user_form_view position=center
;

view user_form_view:
    form for user actions=save update remove view at 3 size 6
;
```

Figure 11. GenAn description of user object and user_form page

For this application, one object for entity *user* was created with properties username (GenAn text type), e-mail (GenAn email type) and gender (GenAn combobox type). Also, a description for *user_form_page* was given. It represents a form for CRUD operations for one user.

```
page home:
    indexPage=True
    title='Home'
    use menubar mainMenu
    use sidebar mainSidebar
    use footer mainFooter
    jumbo:  heading='Welcome'
            text='Lorem ipsum dolor sit amet,
                  consectetur adipiscing elit. Donec interdum ipsum vitae
                  odio congue, in placerat mauris congue. Sed imperdiet sodales finibus.'
            imgPath='http://www.planwallpaper.com/static/images/518079-background-hd.jpg'
            position=top

    paragraph 'Lorem ipsum dolor sit amet,
               consectetur adipiscing elit. Donec interdum ipsum vitae
               odio congue, in placerat mauris congue. Sed imperdiet sodales finibus.
               Lorem ipsum dolor sit amet,
               consectetur adipiscing elit. Donec interdum ipsum vitae
               odio congue, in placerat mauris congue. Sed imperdiet sodales finibus.'
               position=left
    link 'Read more' ref='#' position=left
    paragraph '  ' position=left
    paragraph '  ' position=left

    paragraph 'Lorem ipsum dolor sit amet,
               consectetur adipiscing elit. Donec interdum ipsum vitae
               odio congue, in placerat mauris congue. Sed imperdiet sodales finibus.
               Lorem ipsum dolor sit amet,
               consectetur adipiscing elit. Donec interdum ipsum vitae
               odio congue, in placerat mauris congue. Sed imperdiet sodales finibus.'
                position=center
    link 'Read more' ref='#' class={'style':'float:right;'} position=center
;
```

Figure 12. GenAn description of *home* page

Home page is initial application page. It contains menubar, sidebar and footer (standard GUI components described in Figure 12). *Home* page consists of jumbo *Welcome* panel and two paragraphs. Generated GUI is given in Figures 13 and 14.

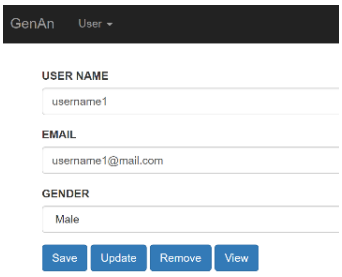

Figure 13. Generated form for *user* object

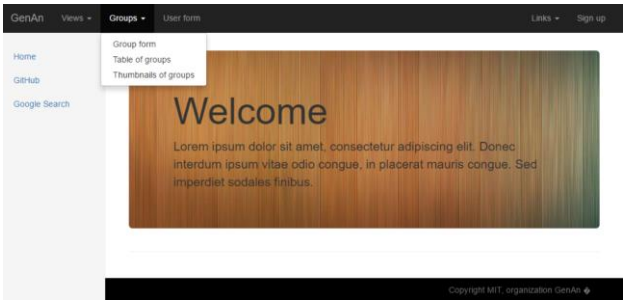

Figure 14. Generated *home* page

V. CONCLUSION

The primary goal of this paper was to present implementation of domain specific language GenAn and code generator for this language. GenAn aimed to provide an uniform and consistent manner of describing graphical user interface for data-oriented web application. Those applications, especially in business domain require very few designer decisions so design aspects should be described in a simple and effortless way. GenAn DSL is created for developers as domain experts and provides fast scaffolding with support for detailed GUI description. The generator is pluggable [24] and allows user to integrate custom code generators to support other platforms. S server and client-side application are generated from the model but using different generators. That decision offered greater flexibility and freedom to end users to re-implement only one of two code generators. Generators are implemented according to existing standards, readable and efficient, with support for validation, localization and query definition.

REFERENCES

[1] Internet Live Stats – Internet Users, http://www.internetlivestats.com/internet-users/, January 2017.

[2] M. Voelter, DSL Ingineering: Designing, Implementing and Using Domain-Specific Languages, 2010-2013.

[3] S. Kelly, J.-P. Tolvanen, Domain Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press, 2008

[4] G. Milosavljević, materijali sa predmeta Metodologije brzog razvoja softvera, Fakultet tehničkih nauka, Novi Sad, 2016

[5] I. Dejanović, G. Milosavljević, R. Vaderna, Arpeggio: A flexible PEG parser for Python, 2015

[6] Zvanična dokumentacija za TextX, http://igordejanovic.net/textX/

[7] MEAN stack, http://mean.io/, January 2017.

[8] GenAn repository, https://github.com/theshammy/GenAn

[9] JHipster, https://jhipster.github.io/, January 2017.

[10] Yeoman, http://yeoman.io/, January 2017.

[11] JDL, https://jhipster.github.io/jdl/, January 2017.

[12] Sifu documentation, https://docs.codesifu.com/, January 2017.

[13] WebDSL, http://webdsl.org/, January 2017.

[14] D.Groenewegen, Z. Hemel, L. Kats, E. Visser, WebDSL: A Domain-Specific Language for Dynamic Web Applications, Delft University of Technology, 2008

[15] Stratego/XT, http://strategoxt.org/, January 2017.

[16] SDF, http://metaborg.org/en/latest/source/langdev/meta/lang/sdf3.html, January 2017.

[17] Spoofax, http://metaborg.org/en/latest/, January 2017.

[18] TextX built-in objects, http://igordejanovic.net/textX/metamodel/#built-in-objects, January 2017.

[19] HTTP, https://tools.ietf.org/html/rfc2616, January 2017.

[20] REST, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, January 2017.

[21] AngularJS, https://angularjs.org/, January 2017.

[22] Node.js, https://nodejs.org/en/, January 2017.

[23] MongoDB, https://www.mongodb.com/, January 2017.

[24] L. Nikolic, B.Zoranovic, I.Dejanovic, G. Milosavljevic, A framework for building component software in Python, ICIST 2017