technische universität
dortmund

Bachelor Thesis

DSL-Driven Generation of User Documentations for Web Applications

Mukendi Mputu
September 2021

Supervisors:
Prof. Dr. Bernhard Steffen
M.Sc. Alexander Bainczyk

Technical University of Dortmund
Department of Computer Science
Chair 5 for Programming Systems (LS-5)
http://ls5-www.cs.tu-dortmund.de

Contents

1 Introduction **1**

1.1 Objectives . . . 2

1.2 Related Work . . . 2

1.3 Outline . . . 3

2 Preliminaries **5**

2.1 End User Documentation . . . 6

2.1.1 Documentation Characteristics . . . 7

2.1.2 Documentation Software Tools . . . 7

2.2 Core Principles of Model-Driven Development . . . 9

2.3 Domain Specific Language . . . 10

2.4 Cinco SCCE Meta Tooling Framework . . . 11

2.4.1 Meta Graph Language . . . 12

2.4.2 Meta Style Language . . . 14

2.4.3 Cinco Product Definition . . . 15

2.4.4 Xtend Generators . . . 15

2.5 Tasks management Web application – TODO-App . . . 16

3 WebDoc - Web Application Documentor **19**

3.1 Graphical DSL . . . 20

3.2 Graph Editor . . . 20

3.3 WebDoc's Model Elements . . . 21

3.3.1 FeatureGraphModel . . . 21

3.3.2 DocGraphModel . . . 23

3.4 Graph Model Checks . . . 25

3.4.1 MCaM Check . . . 26

3.4.2 MCaM Module Check . . . 27

3.5 Generation Process . . . 27

4 Evaluation 31
4.1 Method selection 31
4.2 Setup 32
4.3 Results 32

5 Final Remarks 33
5.1 Conclusion 33
5.2 Future Work 34

List of Figures 37

List of Listings 39

Glossary 41

Abbreviations 43

Bibliography 47

Declaration 47

Chapter 1

Introduction

One of the most challenging tasks in software development is developing a long-term strategy for creating, managing and updating the documentation for the end users as the software product develops, adapts, or increases in complexity [6]. The challenge lies in the fact that the development team tasked to document the software must do it manually by first selecting the information most valuable to the end users, structuring it and then when required updating it. In other words, the developer must reproduce the steps or whole scenarios a potential end user would go trough and document where useful information must be provided to reduce the time needed to understand the main functionality of the software.

Completing this process once does not necessarily free the developer from the task, for it is a cycle that must be repeated every time an update is introduced or a relevant part of the program has been changed. For a small static Web page this might not sound dramatic, but considering complex applications developed by multiple teams, this task can raise the cost in time and resources as the information about the whole project muss be collected for the design of the documentation [1]. So, the problem is to find a way to automatically go through all those steps and generate parts of the documentation model extended with semantic description and later assemble those to generate a complete documentation.

Software projects are bound by budget and deadline requirements. This leads to the fact that automation processes become essential where the reduction of the development time can be increased [15]. It is therefore of significant advantage for the developing team *and* the end user that the generation of user documentation is modeled in such a way that it always reflects the most current development status of the software product [30]. This thesis introduces the use of a graphical domain-specific language (DSL) – a programmatic languages adapted to a specific domain problem [24] – as an alternative way of modeling end user documentation.

1.1 Objectives

This thesis proposes a solution for automatically generating end user documentation for Web application by means of a graphical DSL, a similar approach to [9]. We developed an application for modeling different user action sequences in a Web application using the CINCO SCCE Meta Tooling Framework [12] and subsequently generate an end user documentation in Markdown syntax. The CINCO framework is a generator-driven development environment for domain-specific graphical modeling tools. It is based on the Eclipse Modeling Framework and Graphiti Graphical Tooling Infrastructure, but aims to hide much of their complexity and intricate Application Programming Interface (API)[1]. For evaluation purposes, we document a task management Web application of our own. Nonetheless, the applied method can be extrapolated to any other Web application, since the elementary building blocks of those applications are the same.

The project has a Maven nature, which allows us to manage the package dependencies and take advantage of its build life cycles for building and distributing the application[2].

The Markdown-based documentation is supplemented with screenshots taken with the Web browser automation engine, which drives the Web application to replicate user actions. In addition to that we serve the generated Markdown files containing references to the screenshots as static site.

1.2 Related Work

Documenting a software application – be it as desktop application or a Web application – has always been a daunting task for the developer. Nonetheless, it is a task many developers are neither enthusiastic nor motivated to do [19]. This raised the attention of many researchers on the topic, so that previous similar solution had been proposed in the past:

[14] also proposed in 2015 the Écrit Toolkit, a solution in which the Eclipse Rich Client Platform [16] application model is used to provide a semantic description of the model

[1]More information can be found at https://cinco.scce.info/

[2]The code for the application can be found here https://github.com/MukendiMputu/UserDocGenerator

element. Those semantic description were later aggregated to a user documentation complying with ISO/IEC 26514 [14]. The generation steps consisted of an application model with semantic description being analyzed and converted into an EcritDocument model, that was used as input for a Document Outputter, which produced LaTeXor HTML code. Coding knowledge were necessary to be able to verify the produced documentation. The project is no longer under active development.

GuideAutomator [23] was for example proposed in 2016 as part of a bachelor's thesis. The approach was to provide Markdown files with short JavaScript chunks that determine how to capture each screencast [7]. This implied that preexisting Markdown file were required as input for the generator, which then executed the JavaScript code snippet using Selenium. This approach limited itself in a way that knowledge of the JavaScript programming language where required and that the Markdown file had to manually be generated prior to the execution the generator, which did not support reusability of such Markdown file. According to the last GitHub commit, which goes back to September 2018, to project is not being actively developed any longer.

In this thesis we decided to keep the philosophy adopted by the developers of the Cinco framework. Our approach prioritizes reusability and simplicity of our graphical language amongst the most important features we aimed to implement. This enables non-programers to design graph model of the Web application they desire to document just by arranging the graphical elements our language proposes to a logical sequence of actions. The generation of the executable application code and all the documentation files necessary is taken care of by the editor. We also assist the documentation designer by checking the syntactical correctness of the diagram at runtime. No prior knowledge of any programming language is needed throughout the complete design process.

1.3 Outline

Chapter presents the Cinco SCCE Meta Tooling Framework, in particular the two meta-languages (Meta Graph Language (MGL) and Meta Style Language (MSL)) that constitute the meta-specification (Section 2.4.1 and 2.4.2) of our graphical domain-specific language. The chapter then rounds off with an outline of the Generator classes written in Xtend, a Java dialect that offers much flexibility and expressiveness in its syntax (Section 2.4.4).
The specifics of the generated modeling platform are described in chapter 3. Whereby Section 3.1 explains the basic concepts of a graphical domain-specific language, then Sections 3.3.1, 3.3.2 and 3.5 relate about different modeling elements for configuring and designing the end user documentation. In chapter 4 a in-depth explanation of the methodology is

given. Beginning with the evaluation of the selected work method in Section 4.1 followed by the system setups and the result of the approach taken in this thesis(Section 4.2 and 4.3). Further improvements of the program are presented in chapter **??**. Lastly, chapter **??** winds up with a discussion on the thesis.

Chapter 2

Preliminaries

This chapter introduces the key concepts that are required to achieve our goal. We begin by considering the format of our end-user documentation as required by known standards, as well as the technologies available to assist us in completing this task. The CINCO Meta Tooling Suite will next be used to introduce the principles of model-driven programming, as well as introduce the meta languages on which our graphical models are based and quickly describe what needs to be generated from those models. The steps to achieving our aim are depicted in Figure 2.1.

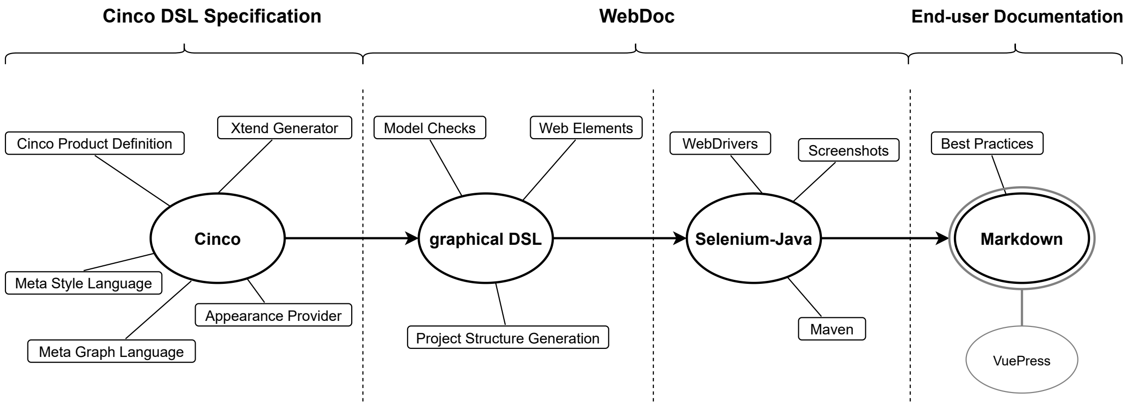


Figure 2.1: Development path of the end user documentation

2.1 End User Documentation

The value of a good software product, in fact, of any product destined to be brought to the consumer is determined on how effective the end user can learn to use the product. Hence, putting a great effort to generate and manage a useful, well-structured documentation is as much important as the development of the product itself [3].

The goal of end user documentation is to provide application end users with the knowledge they need to interact with the software properly. It is an integral aspect of the development process and serves as a link between the product creator and the end user [4]. By increasing the structure and usability of the information supplied to end users, the developer not only reduces the number of support calls, but also improves the product's and the company's reputation [1].

In this section we will go through the main characteristics of end user documentation, focusing on the standards establish by the ISO/IEC and the IEEE [1, 3]. Next, we will talk about the specifics of documenting Web application by giving an example of a documented Web application. We will also present the currently used technologies for creating and managing such documentation. Our main goal is to produce an entire end user documentation page from a graphical model. Before going further, we ought to precise that the type of documentation to be created will depend on how the documentation designer lays out the model, that means the model could be designed to represent a technical documentation – requiring deep knowledge of the application documented – or it could also be structured to produce a documentation for simple end users with no technical knowledge at all of the underlying Web application [6].

2.1.1 Documentation Characteristics

Since our focus is primarily on Web applications, we will be documenting the User Interface (UI), which is composed of Web elements (like buttons, navigation links, checkboxes, etc.) the user can interact with. This settles implicitly the type of documentation we aim to create: a step-by-step guide for navigating UI to reach the expected result. Nonetheless, the ISO/IEC/IEEE Standard 26511:2012 [2] mandates i.e., that completeness and accuracy should be of the utmost importance when designing the end user documentation.

Bearing that in mind, a good documentation should respond to three important questions: *why* the application has been programmed, meaning the problem the application intends to solve, *what* is the end user supposed to do to get to the solution quickly and efficiently and *how* it is achieved [3]. This is a task we leave to the information manager. The last question, however, is better answered by providing visual help in form of screen captures or any illustration that fastens the understanding of the documentation.

We plan to document Web applications from the perspective of the application's end user, as previously stated. This means that the documentation developer must consider the product's audience, and identifying the core jobs and activities of common users should be an important element of the design process [3]. The developer is in command of how extensive the documentation becomes using our solution.

2.1.2 Documentation Software Tools

Choosing the format in which the documentation is brought to the end user is as much important as the rest of the milestones of the planning process. A poor decision in this area might limit the documentation's ability to reach a large number of consumers, rendering it useless. Choosing technologies that have already received widespread adoption is a wise move. This ensures a wider reach and prior knowledge within the same audience:

Markdown

Markdown is a lightweight markup language created by John Gruber[1] in 2004. Since, it has established itself as one of the world's most popular markup language [13]. Gruber states on his webpage that Markdown is comprised of formatting syntaxes that can be applied to plain-text file and optionally a converter to other markup languages files like Hypertext Markup Language (HTML). In this acronym, Hypertext means the link that connects Web

[1] John Gruber's official project website https://daringfireball.net/projects/Markdown/

pages to each other, either within the same website or between different websites [20]. On Mozilla Developer Network website[2], HTML is described as the fundamental building block of any Web page, that uses markup syntax to annotate text, images, and other content for display in a Web browser.

For its popularity, it is a great choice for our project, since description of the UI is first made as plain-text and then transform to a rich-text format using markup syntax as depicted below. Moreover, Markdown files can be opened and edited with any kind of text editor available. There is tremendous amount of documentation about Markdown syntax available online for the interested reader to get started.

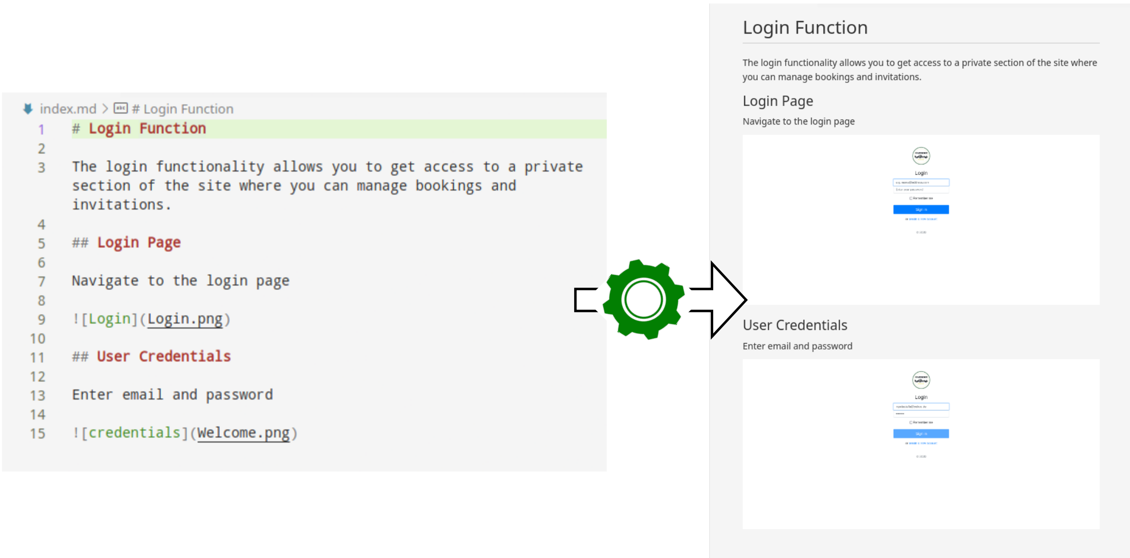


Figure 2.2: Plain-text transformation with Markdown

Although there exists a plethora of applications that transform Markdown files into HTML files to be rendered in a Web browser application (i.e. MacDown [1] for Mac users, ghostwriter [2] for Windows and Linux or online tool like Dillinger [3]), we choose to work with one framework that also established itself throughout the developer community and that especially works very well with Markdown files: VuePress.

VuePress

VuePress is a minimalistic VueJS-powered static website generator. VueJS [31] is an open source JavaScript framework designed by Evan You[4]. It is called minimalistic because it

[2]MDN: https://developer.mozilla.org/en-US/docs/Web/HTML
[1]MacDowns: https://macdown.uranusjr.com/
[2]Ghostwriters: https://wereturtle.github.io/ghostwriter/
[3]Dillinger : https://dillinger.io/
[4]Eva You's website https://evanyou.me/

only takes a few lines of configuration and two commands to create and launch a static website. It is resilient and up-to-date since it is open source software and is maintained by a large community of contributors.

By respecting the VuePress conventional folder structure, it can explore folders, get Markdown files, and convert them to HTML files. The default appearance of the generated website is defined by a customizable configuration file included in the project folders. By simply adding certain configuration elements, you may create i.e. a sidebar menu and a navigation bar. VuePress produces a static website by building hyperlinks between HTML files in the project folder hierarchy in the order in which they appear. This means that subfolder Markdown (or HTML) files are rendered as menu items on the folder's HTML page. A server-rendered website with a live reload mechanism is constructed and launched after inputting the correct command. Every modification to the Markdown files is immediately visible on the website. The final step is to create the website, which will be saved in a special folder and can subsequently be exported to any other website.

We will use VuePress to statically launch the end user documentation as a website, relieving the developer of the task of configuring the server. The principal task remaining is therefore to create a solid documentation model.

2.2 Core Principles of Model-Driven Development

This section lays down the fundamentals of the model-driven development (MDD) – also referred to as model-driven software development (MDSD) [18] – using the CINCO SCCE Meta Tooling Framework, as well as the steps necessary to get up and running with the framework. The term MDD refers to a development paradigm where the functionalities of the software are first specified as models, from which then executable code can be automatically generated. To create and use models to represent the software system, a DSL that is close to the problem domain, is needed. Application domain experts possess the knowledge of the application structure, which they represent in form of models. Domain experts and application programmers must agree on how the specification language determines syntax and semantic of the model elements.

The core principles of MDD reside in the fact that software development is accelerated by providing a simple, but efficient abstraction of the software structure as a model. Those model abstraction, representation of real-world objects, are transposed through a series of model-to-model or model-to-code transformations [27]. Our work is to utilize the DSL provide by the CINCO framework to design our graphical DSL, which in turn will permit the generation of a functioning Selenium-Java application (Selenium is a suite of application tools for automating Web browsers) to take screenshots of the different Web application states.

Opposed to the common development method, applying a graphical model to layout the different user sequences allows even non-programmer (here the domain expert with much more expertise on how to design a great software documentation) to accomplish the task of documenting the features offered by the Web application. Nonetheless, the programmer has the tasks – in collaboration with the domain expert – to specify the meaning of each model element for the code generation process.

When applied correctly, the result of the model-driven development process is a tailored application to domain. This reflects one of the main advantages of MDD: the accuracy of directly targeting the specific problem [10]. Besides, it is still possible to change the DSL so that it adapts to the new challenges emerging during the development process. This can be iterated until the specification reaches preciseness wanted to solve the problem.

2.3 Domain Specific Language

A domain-specific language (DSL), as mentioned before, is a language adapted to specific development domain. In [22] a succinct analogy to DSLs is given by saying that it is comparable to a tool specially crafted only for one specific task as opposed to general programming languages, which can be seen as tools for multiple different tasks. If we stick to this analogy, just as one would start with a blueprint to construction a mechanical tool, designing as DSL required similar steps. One must conceptually lay out the behavior and eventually – in case it is a graphical language we seek to design – the look of each element that can be used in our DSL.

Blueprinting our DSL is equivalent to defining a metalanguage or meta-DSL to our language. *Meta* literally means *situated behind or beyond*[5]. So, metalanguage is a descriptive language that comes before our language and describes the meaning (semantic) and relationship between the objects of our target language. To put it another way, the meta-DSL is the abstract syntax, and the DSL concrete syntax is the result. It is feasible to work our way up the meta-definition modeling hierarchy until we reach a self-referencing language, like it is the case Unified Modeling Language (UML).

In our case, we want a graphical language for creating models that represent the different parts of a Web application and how the end user can possibly interact with them. The metalanguage to our graphical DSL is provided by the CINCO SCCE Meta

Tooling Suite. It comprises the Meta Graph Language (MGL), the Meta Style Language (MSL) and the Cinco Product Definition (CPD). All have been constructed using Xtext [8], a language Workbench for writing textual DSLs [21]. The next section explains those concepts in detail.

2.4 Cinco SCCE Meta Tooling Framework

The Cinco Framework is a generator-driven development environment for domain-specific graphical modeling tools [12]. It is actually developed by the chair of programming systems at the Technical University of Dortmund and one of the many projects of the SCCE Group [1], which aims at allowing the application-domain experts, rather than programming experts, to take charge of the development tasks [25]. One of the great features of this framework is that it allows us to generate an entire editor application with just one click from a simple textual specification language – the MGL mentioned in the previous section.

The MGL together with the MSL form the metamodel from which Cinco generates a ready-to-run modeling tools called Cinco product. The MSL is where the Cinco developer defines the look every node and edge element, as well as the font and color of the text to be displayed in the graphical model [21]. The created metamodel is based on Ecore, the metamodeling language of the Eclipse Modeling Framework (EMF) and the Graphiti framework is used to generate the corresponding graphical model editor. Additionally, you find the right button to trigger code generation in the created editor. Chapter 3 is devoted to our Cinco product (the editor application), in particular section 3.5 gives an in-depth explanation of the generation process.

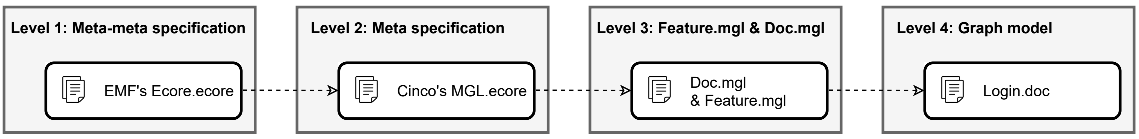


Figure 2.3: Hierarchy of our graphical DSL specification

The full generation process of a graphical modeling tool (of a Cinco Product Application) comprises four essential (meta) levels [22]. The first level, associated with the role Eclipse Developer, is where the Ecore.ecore and GraphitiDiagram.ecore metamodel are developed. The second level is where Cinco Developers of the Chair 5 for Programming Systems used the metamodel from the first level to develop the MGL.ecore, which in turn becomes metamodel of the third specification level, where the Cinco Product Developer

[1] https://www.scce.info/

operates and where this thesis comes in, see fig. 2.3. This means the Eclipse Developers are now at the meta-metalevel, the Cinco Developers at metalevel of the specification elaborated in this work. For the remainder of the chapters we will be focusing on the third and fourth specification level. Latter is the level where the Cinco Product Users use the generated graphical editor to create the domain specific models.

2.4.1 Meta Graph Language

The MGL sketches the behavior, the constraints and gathers the all the graphical components that will constitute the model elements that come in use in every graph diagram created in the modeling tool. This is where we start developing our editor application: the main elements that can be define in a MGL are **nodes**, **containers** and **edges** as illustrates listing 2.2.

The main graph model, the `FeatureGraphModel`, is the one containing all the features of the Web application we want documented. For instance, our Web application provides the login feature as starting point, then after successfully logging in and getting access to the dashboard, task lists can be created or deleted, tasks can be added to or removed from them, etc. The graph diagram can be assigned name and description. We specified `.feat` (short for feature) as model extension. And since the login feature i.e., requires a series of user actions to be performed, the FeatureGraphModel contains another graph model type, where those sequences of actions will be modeled: the `DocGraphModel`. It contains the meta specification of all the commonly used Web elements (i.e., input field, buttons, dropdown, etc.) we want to use in our graph models, as well as sectioning elements to structure the diagram and edges to connect them to each other. The listing below shows an excerpt from the concrete FeatureGraphModel implementation.

```
graphModel FeatureGraphModel {
    iconPath "icons/16/feature_16.png"
    diagramExtension "feat"
    containableElements(FeatureContainer,DocNode[1,*])
    attr EString as modelName := "New Feature"
    attr EString as description := "New Description"
}

container FeatureContainer {
    style featureContainer("${title}")
    containableElements(Start[1,1], Stop[1,1],DocNode[1,*])
    attr EString as title := "New Feature"
    @multiLine
    attr EString as documentation := "This feature is about ..."
}

node DocNode{
    style docNode("${mgl.modelName}")
    prime docMgl::DocGraphModel as mgl
```

```
20     attr EBoolean as createScreenshots := true
21     incomingEdges (Edge[1,1])
22     outgoingEdges (Edge[1,1])
23 }
```

Listing 2.1: Excerpt from the feature.mgl, meta-specification of the FeatureGraphModel

As you can see we define a container element to hold the DocGraphModel instances (see lines 9 and 17). To integrate a whole DocGraphModel inside a FeatureGraphModel we used the so called *PrimeReference*, which is a Cinco feature that allows to reference another model by only one attribute of a node or container [12]. It is applied by adding the keyword `prime`, as we did in line 9. By doing so, we can simply drag and drop a DocGraphModel diagram into a FeatureGraphModel diagram to incorporate it.

As for the DocGraphModel, we specified four categories of model elements to use while designing the diagram: the most important ones, the Web elements, represent the UI element the user can interact with. The we have the Selenium action nodes, which perform some actions to drive the Web browser through the Selenium WebDriver. This are i.e., the Navigation node to change from one Web page to another and the Screenshot node to capture the application current state as an image. Finally, we have the semantic element `Comment`, to give a descriptive text to the screenshots and the basic elements (start and end as well as the section node) which help structure the user sequence graph.

```
graphModel DocGraphModel {
    iconPath "icons/16/sequence_16.png"
    diagramExtension "doc"
    containableElements(*)
    attr EString as modelName := "UserSequence"
    @multiLine
    attr EString as documenation := "Lorem ipsum dolor et si met"
}

node Screenshot {
    style screenshotNode
    incomingEdges (Transition[1,1], Anchor[1,*])
    outgoingEdges (Transition[1,*])
    attr EString as pictureName
    attr Comment as description
}

node Input extends WebElement {
    style inputNode("Input: ${content}")
    attr EString as content
    incomingEdges (Transition[0,*])
    outgoingEdges (Transition[0,*])
}
```

Listing 2.2: Excerpt from the Doc.mgl, meta-specification of the DocGraphModel

For demonstration purposes, we kept our example listings short. An exhaustive list of all the usable elements and annotations can be found on the Cinco's documentation page[1].

2.4.2 Meta Style Language

The appearance of all the elements defined in the MGL are laid down using the textual Meta Style Language (MSL). As explained in [11], three essential elements constitute the design of a MSL model, namely: **appearance**, **nodeStyle** and the **edgeStyle**.

Each nodeStyle specification makes use of an appearance element, which in fact determines the attributes like background color, the thickness of the drawn lines and so on (see listing 2.3). The nodeStyle is hierarchically composed of a shape that can be given a `size`, `position` and a `text` element with a `value` attribute that takes a (format) string (line 12) that will be display in the graphical model. We can say that it is left to the Cinco product developer's imagination to style the elements as seen fit.

```
1  nodeStyle featureContainer(1) {
2      rectangle {
3          appearance extends default {
4              background (246,245,244)
5          }
6          size (300,75)
7          text {
8              appearance {
9                  font("Sans", BOLD, 10)
10             }
11             position (LEFT 5, TOP 5)
12             value "%s"
13         }
14     }
15 }
```

Listing 2.3: Excerpt from feature.style to be applied to feature.mgl

It is also worth mentioning that the concept of inheritance from the Object Oriented Programming (OOP) can be applied between metamodel element of the same type, hence avoiding repetitive definition of the same attributes within multiple different elements, and allowing some elements to extend the properties of the parent elements. For example, we see in line 3 the `featureContainer` appearance extends the default one and at the same time redefines the background color.

[1]Wiki page : https://gitlab.com/scce/cinco/-/wikis/Cinco-Product-Specification

2.4.3 Cinco Product Definition

In the Cinco Product Definition (CPD) is where it all comes together. Herein, the Cinco Product Developer must provide key information like the Cinco product name, at least one or more MGL files to be included into the generation process. Optionally, one can setup a splash screen with branding images, add a descriptive text about the application and specify plugins and/or features [11]. The listing below gives an insight into the CPD specification language.

```
CincoProduct UserDocumentationTool {
    mgl "model/Feature.mgl"
    mgl "model/Doc.mgl"

    splashScreen "branding/splash.bmp" {
        progressBar (37,268,190,10)
        progressMessage (37,280,190,18)
    }

    image16 "branding/Icon16_dark.png"
    image32 "branding/Icon32.png"
    image48 "branding/Icon48.png"
    image64 "branding/Icon64.png"
    image128 "branding/Icon128.png"
    linuxIcon "branding/Icon512.xpm"

    about {
        text "WebDoc is a DSL-driven generator of end user documentation
for Web application. It is a bachelor thesis project developed with the
Cinco SCCE Meta Tooling Suite ( http://cinco.scce.info )."
    }

    plugins {
        info.scce.cinco.product.userdocumentation.edit,
        info.scce.cinco.product.userdocumentation.editor
    }
}
```

Listing 2.4: UserDocumentationTool.cpd

2.4.4 Xtend Generators

For our documentation application we need to create two different application folder structures from our model diagrams: the first one is the Selenium-Java application, that basically replays the modeled user action sequences and takes screenshots as laid out by the designer. It follows the specific Maven project structure, for which we applied the rule of convention over configuration as recommended on the Maven Apache [1] website and added Selenium as a dependency. The other project structure we generate

[1] https://www.apache.org/

by following convention is for the VuePress project, which in fact is the result we aim to obtain. In this project, we generate the Markdown files containing all the semantic text the documentation developer specified as description and/or comment in the model elements. This is achieved by following each sequence beginning from the start node all the way through to the end note, constructing a cohesive documentation text.

This generation approach utilizes Java's and Xtend's text templating feature which is based on the generation pattern used in the Java Application Building Center (jABC) [28, 29]. In fact, many generator classes in our example project implement and extends interfaces from the generator runtime and template package of the jABC CINCO meta plugin.

Additionally, other template classes are implemented to create configuration files, application class and project files. In chapter 1 section 1.1 we provided a link to the GitHub repository, where the complete application code can be found. In the next chapters, we show the output of the generator and feature classes, while introducing our application as an ongoing example.

2.5 Tasks management Web application – TODO-App

The TODO-App is a Web application fully generated with the DyWA Integrated Modeling Environment (DIME) [9], a development environment for creating Web applications. DyWA stands for dynamic Web applications, it is the container which is used to host the generated application. While modeling and code generation happens in DIME, DyWA provides support for the product deployment phase, constitutes the runtime environment, and manages the data persistence [9].

The TODO-App is an application that helps management lists of tasks that need to be done. It provides to a logged-in user with the possibility to create such lists, add various tasks to them, and after their completion, remove them [9]. Users also can add co-owners, which are other existing users, that can manage that respective list with them. The application is launched in development mode, meaning that it is accessible via a local Web address (`http://localhost:8080`) from the host it has been started in. The TODO-App is ideal for demonstration purposes, since it presents all the Web elements necessary to a Web application interface. Figure 2.4 shows said interface with all the elements the user can interact with, i.e. buttons, input fields, dropdown selectboxes, etc.

DIME's model-driven development concept was our starting point. That is, the graphical elements meta specification is focused on specifying elements that represent the

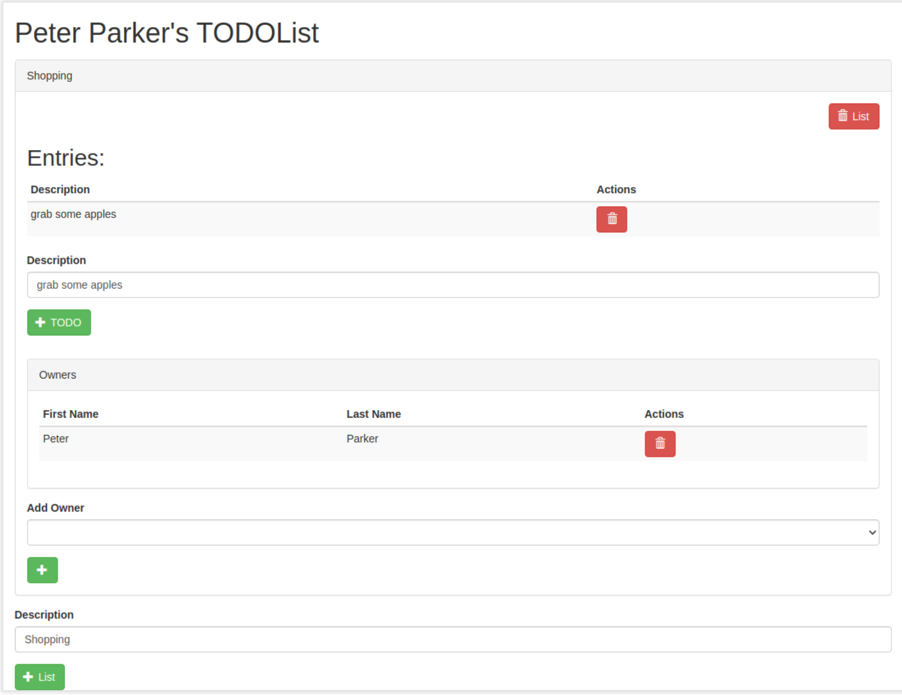


Figure 2.4: Impression of the TODO-App Web interface

graphical user interface (GUI) and those that implement processes. The latter are elements whose semantic implementation will direct the Web browser to recreate the procedures necessary to construct the intended documentation. However, we should point out that in our scenario, the concept of data permanence is not required. Furthermore, even though our concept is based on DIME, it does not mean that our application is only for DIME-generated Web apps. In this respect, any navigable Web application can be modeled with our editor and thus documented.

Chapter 3

WebDoc - Web Application Documentor

We've gone through the CINCO DSL specification for the graphical modeling tool, and now we'll go over the model editor that was created from it. We begin with a brief description of our graphical DSL, then demonstrate the key building elements of our documentation architecture while simultaneously demonstrating our CINCO product: WebDoc[1]. We'll show how to use these graphical elements to simulate a specific user work-flow on the website we're documenting later. Finally, we demonstrate how to construct the target application that generates the Markdown files that will be shown in the Web browser using the editor's built-in generator.

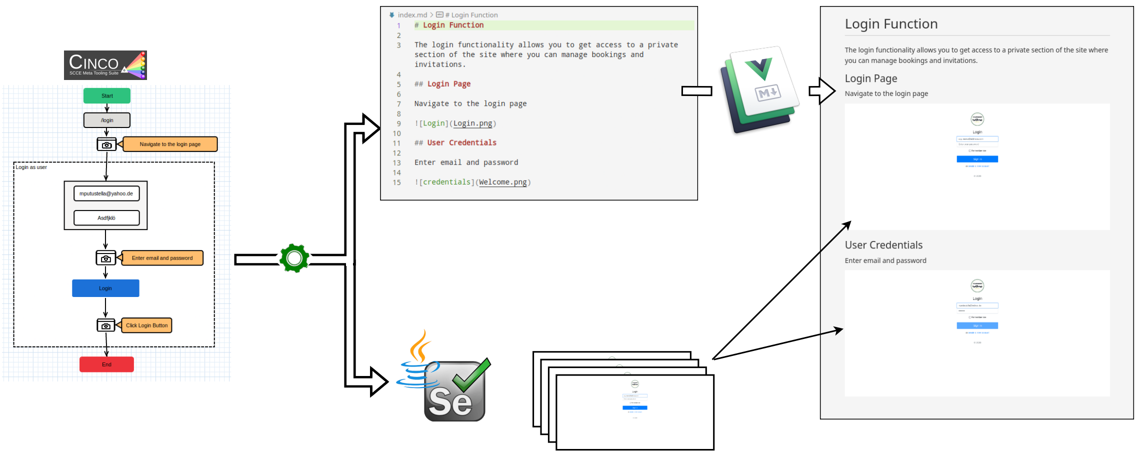


Figure 3.1: Generation process of End user Documentation

[1]GitHub repository: https://github.com/MukendiMputu/UserDoc

3.1 Graphical DSL

Our graphical DSL mainly comprises node elements, which have been applied different appearances to, must resemble to some extent the corresponding Web elements they represent, as well as connector elements to connect the nodes to sequence graphs. The purpose behind this approximated replication is not to recreate all possible Web elements, but to give the developer a sense of control over the interactable UI elements. With those elements at hand, the designer can simply model a user action by arranging node elements following the navigation path of the Web application. This graphical language is in such way domain specific, that it is tailored for modeling Web applications inside a Web browser; modeling a documentation for a desktop application for example would not be possible.

3.2 Graph Editor

The WebDoc editor is mainly composed of the canvas in the middle of the working environment (1), where the developer can drag and drop model elements from the palette located on the right-hand side (2). Herein, elements are grouped in categories. On the left-hand side we have the project explorer showing the current project structure (3). The main model files have the extensions .doc and .feat, where the latter is the entry point of the documentation application.

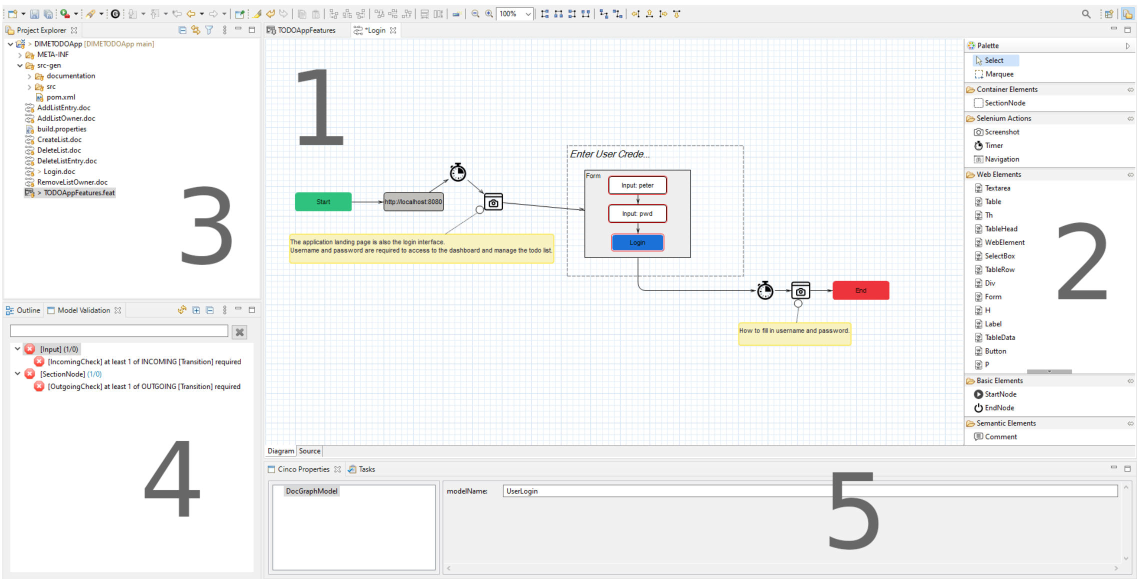


Figure 3.2: Cinco Product Application - Graph model editor

All diagrams that are open in the editor are checked in the background for compliance with the requirements described on the metalevel and shown in the Model Checking view right beneath the project explorer (4). There is also the Cinco property view (5) that displays the attributes and values of any selected element in the editor. This is where the developer can modify those values if the attribute field allows it.

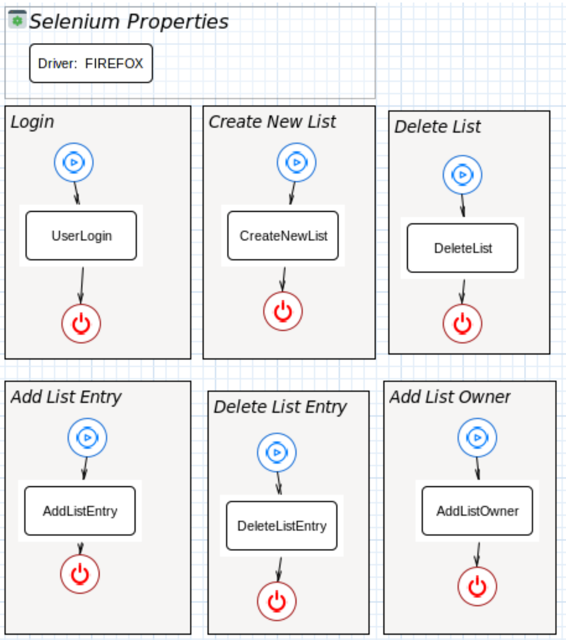
3.3 WebDoc's Model Elements

On a Web page rendered in the Web browser, there is a small amount of HTML elements the user can interact with. Most of them resides within the HTML forms, tools that are used for collecting data from the user or allowing them to control a user interface [20]. Input fields (of all types: text, password, textarea, checkbox, etc.) and buttons are the prominent ones. Those are the Web elements that primarily constitute the palette of our model elements. In the previous chapter, we defined two different MGLs: one for modeling the Web application features and the other one for modeling the user actions that make up those features.

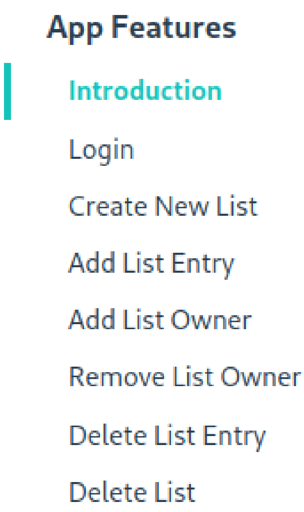
3.3.1 FeatureGraphModel

The FeatureGraphModel is the application starting point specified by the `feature.mgl`. Here, the developer groups all the features that needs to be documented in feature containers. Figure 3.3a shows a sample of the features modeled for our task management application, including the ability to login, create a new list, add or remove a task from that list, and finally delete the entire list. The documentation website's resulting sidebar is depicted in Figure 3.3b, albeit it should be noted that the "Introduction" is not an application feature, but rather the introductory page displaying the various functionalities. Moreover, our application offers the possibility to also add a new list owner, which already exist in the system as regular user.

On the top left-hand corner, you can see an property container holding the WebDriver property, whose value is set to `FIREFOX`. It is one of the additional model elements that help the developer define configuration that otherwise could not be modeled, but still essential for the execution of the application. This property value will be assigned to the Selenium WebDriver variable in the Java class. Remember that the executable file for the chosen WebDriver must already exist somewhere in file system and the path to it must be specified here in the property view.



(a) Features in the FeatureGraphModel



(b) Menu structure on the documentation website

Despite the fact that the features are provided in a logical workflow order, they are still distinct from one another and will be generated separately. As indicated in the last chapter, however, this separation does not preclude reusability, as it is feasible to embed one whole DocGraphModel within another. Consider the CreateNewList functionality, which necessitates the user being signed in in order to create a new task list. So, instead of repeating the login sequence in this one, the documentation developer may simply drag and drop the UserLogin.doc file into the diagram of the new model graph and connect it to the sequence as if it were a regular graph node. (see figure 3.4). When you double-click on an imported subgraph, you'll be sent straight to the original graph model. This double-click action was implemented by annotating the associated node specification with the `@doubleClickAction` annotation and providing a custom action class that holds the implementation logic. Furthermore, by unchecking the createScreenshots checkbox in the property view of a subgraph, the designer can disable the creation of screenshots for that specific model, which is enabled by default. This prevents the same screenshot from being taken several times when the subgraph is reused.

The semantic elements in the feature graph model are incorporated into the underlying featureContainers for simplicity's sake. When you click on a featureContainer like this, it opens the Cinco property view, which includes a multiline input form called 'description.' This is where the documentation designer can offer text content for the Markdown documentation files while also providing descriptive text for the diagram objects. These

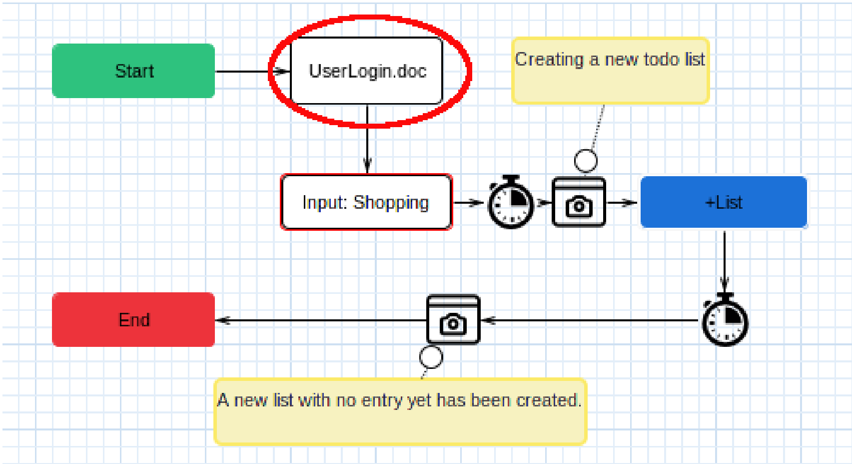


Figure 3.4: Cinco product - Reusing Login graph model in the CreateNewList graph model

information texts will be gathered during the code creation process to create the complete documentation text.

3.3.2 DocGraphModel

Figure 3.5 illustrates the steps a user would take to log into our example Web application, as well as the Web elements that will be interacted with. They create the user login sequence for our Web application when connected in a logical sequence.

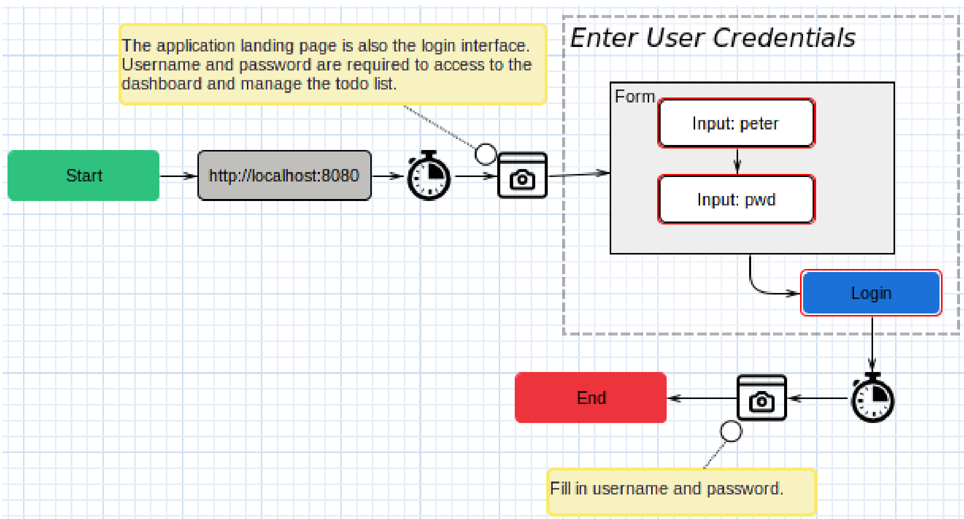


Figure 3.5: Example of a user workflow: here the login sequence

The sequence begins with the start node, which is the starting point of every user sequence, then comes the navigation node, used to navigate from one Web page to another. Next, the timer node waits explicitly for an amount of second determined by the designer in the property view and for a certain condition to become true before continuing. Those ExpectedConditions are i.e., `presenceOfElementLocated`, `elementToBeClickable`, just to name a few. If we take for example the condition `presenceOfElementLocated`, the timer holds the WebDriver execution for 3 seconds, checking every 500 milliseconds if the targeted Web element appears on the page. This is a necessary step to capture all elements of the page while taking the screen capture, as some elements take time to be completely loaded. Right after the timer comes the Screenshot node, that captures the current application state, displaying the application landing page, which also is the login pane. In the property view, one can give a unique file name to each image to be saved. The comment node allows the documentation creator to add descriptive text about the picture, that will be later added to the Markdown file as image caption. Next comes the Section node, that regroups form elements for entering and validating user credentials. As you can see, all input nodes, as well as the button node, have a red border that represents the highlighted state of the element. The model designer may see which elements will be highlighted in the following screenshot, after which the login sequence will end.

The purpose of the Web element nodes is to allow the concrete HTML elements in the browser to be addressed. This implies that appropriate actions can be applied to such a representation of an HTML element. Considering for instance the input field in the picture below, the action of typing in a text is made possible by providing a property variable `content`, whose value is then display inside the node element (here i.e. "peter" and "pwd"). The same holds true for button elements, which can be applied the click action or for selectboxes, which can be dropped down to reveal the options they contain, etc. In addition to that, all Web elements can be highlighted either by setting the `highlighted` property to true or by letting the Highlight node under the *Selenium Actions* category take care of it. The screenshot taken immediately after will have that specific element surrounded with a red border as well (see fig. 3.6).

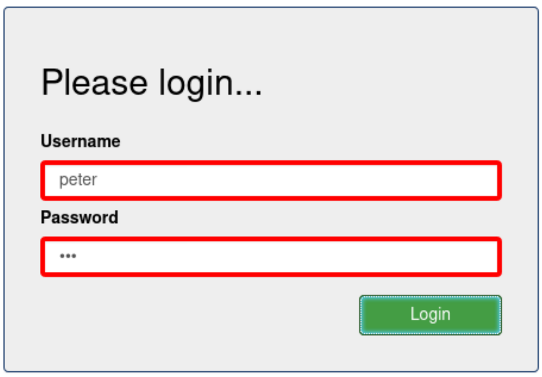


Figure 3.6: Login pane with screenshot of the highlighted input element

In addition, each node element includes a description attribute, which, as discussed in the preceding section, provides additional content for Markdown files. We decided to hide this attribute in the property view to avoid overcrowding the diagram with semantic components describing each and every node member. All model diagrams are traversed during the generation process, and all descriptive texts are collected into paragraphs, then sections, and finally full Markdown files, depending on the feature they are included in. As a result, the documentation content is organized in the following fashion (see fig. 3.7): the featureContainer title is rendered as page headings, the DocGraphModel name is displayed as section headings with all other element descriptions underneath it, and so on.

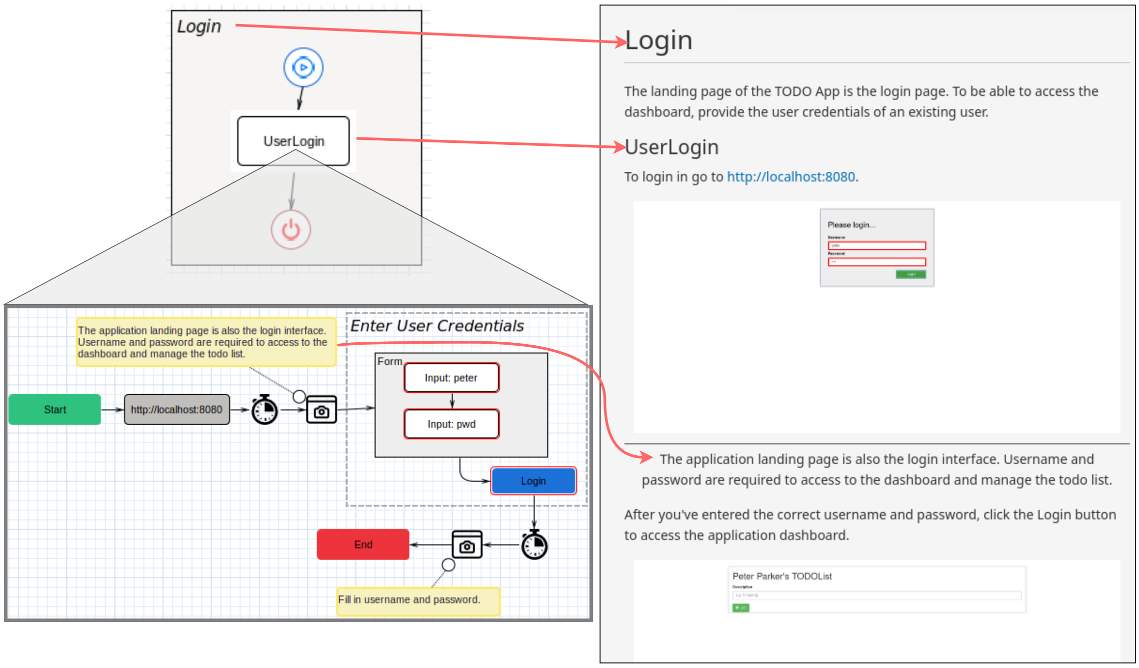


Figure 3.7: Structural mapping of models to Markdown

3.4 Graph Model Checks

Modeling valid graph diagrams is crucial for the correctness of the generated application code. Especially the Selenium-Java application code must correctly replicate the user sequence in order capture the correct application state as the user would do. This is particularly daunting if the navigation graph of the underlying Web application is huge.

This means that we need to check that two graph model do not have the same name or that for each user activity, there is a distinct path and that there is no cycle inside that path. This is reminiscent of various graph theory problems, such as determining whether a path from the start to the end node exists and whether a graph is cycle free. The MCaM plugin, which is incorporated into the Cinco framework, can be used to do this operation [11]. The plugin must be activated by specifying the `@mcam("...")` or the `@mcam_checkmodule("...")` annotation with the corresponding parameter in parentheses. Also, nodes and model graph attributes can be specified with the **`unique`** qualifier to enforce the uniqueness of the model name at design time.

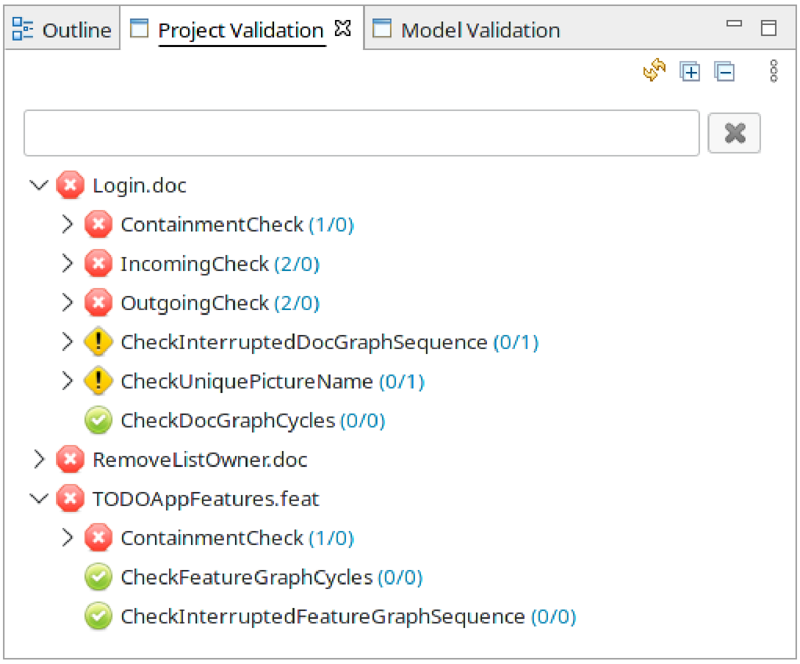


Figure 3.8: Project Validation view showing passed and failed checks

3.4.1 MCaM Check

The first one must be declared right above the **`graphModel`** declaration with the parameter "check". It activates the check of the constraints on the graph nodes in the graph model scope. At design time the model creator receives feedback in form of errors, because those mistakes either cause the generator to throw an runtime exception or the resulting application code will be error-prone, hence, not executable.

For this check to show a successful Incoming- or OutgoingCheck, the multiplicity of the relationship between node elements must respect the specified value. For example, a node element specified with an incoming Edge in this fashion **`incomingEdges`** `(Edge[1,*])`, must have at least one incoming connection of the `Edge` type. Any other kind of multiplicity can be defined; even not specifying any implies one. For instance, writing **`incomingEdges`** `(Edge)` is equivalent to **`incomingEdges`** `(Edge[*,*])`, which means at least and at most any number of `Edge` transition.

When specifying the containable elements of a container node (see i.e. line 11 in listing 2.1), with the `containableElements` property, each elements listed in that property has a one type of multiplicity explained above. If not respected, the validation view will show a ContainmentCheck error until it is resolved.

3.4.2 MCaM Module Check

The second annotation allows us to specify our own module checks. For instance, we need to ensure that two different screenshot nodes do not bear the same file name or that each sequence starts with the appropriate StartNode and ends with the EndNode. The former check makes sure that no previous screenshot be overwritten by another one with the same name and latter ensures that the resulting code reflects the model sequence from start to the end. We also need to ensure that by integrating a DocGraphModel into another one we do not create cycles within the execution path, otherwise this would result into a never-ending execution loop once the generation process has been triggered.

This module check allows us also i.e. to check for unique feature in the FeatureGraphModel scope. Although the independency of each feature is ensure by the featureContainers, two featureContainers bearing the same name would result in two documentation pages having the same title. The *Project Validation* view shows all the checks done on the entire project and marks those that failed or passed the checks accordingly. Figure 3.8 shows for example the checks done on our example project, where the Login.doc DocGraphModel passed the cycle check, but fails the one checking for interruption and unique picture names.

3.5 Generation Process

On the design process of all required models is complete, the executable code is now to be generated. Recall that we want to generate two different projects: the Selenium-Java project to capture the screenshots and the VuePress project containing all the documentation Markdown files. We thought of two ways to reach the intended result.

The first one consist of generating the Selenium-Java application from our model graphs and subsequently, when executing generated application, create the VuePress project with all the documentation files as shown in fig. 3.9. This approach would imply letting Java take care of the generation process of the Markdown files as well as the creation of the needed screenshot. Erroneous generation of the Selenium-Java application will leave the designer with no palpable result at all. In this approach generated application becomes a single point of failure.

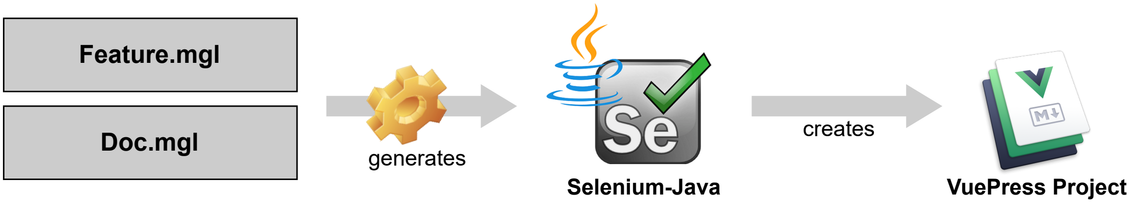


Figure 3.9: First approach: Sequential generation of the documentation

The second approach applies the concept of separation of concerns (see fig. 3.10). That means that we separate the task of taking screen captures with Selenium-Java from the one of generating the documentation files that will contain those screen captures. This way, if the Selenium-Java application contains errors that prevent from taking screenshots, we still can generate a documentation website with placeholder pictures, that can be later replaced with manually take screen captures. That being said, it is almost clear now that we opted for the second approach to create the documentation website.

So, clicking on the generate button creates a executable Selenium-Java application, which, once started, runs the user sequence model as a Selenium script in the Web browser. As mentioned in the preliminary chapter, the generator classes behind this process are written in Xtend, as statically-typed programming language based on Java and translates to Java source code [17].

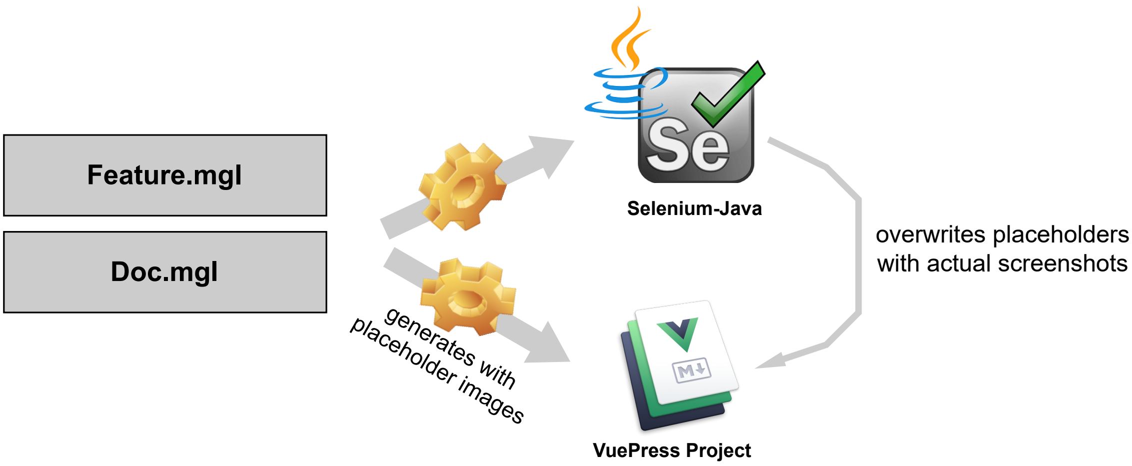


Figure 3.10: Second approach: Separate generation of the documentation

To be able to start a generation process from a graph model we must declare it ”generatable”. We do so bei adding the `@generatable("path.to.generator.Class")` annotation in the graph model meta-specification. This annotation accepts as parameter a Xtend or Java class that implements the IGenerator, whose `generate` method starts the whole process and the path to a location where the generation product will be saved (in our case it is in the `src-gen` folder). Recalling the goal we set for application code, we create two folder structures inside the src-gen folder: the first one is the Java project structure with the Selenium script and the second one is the VuePress project holding all the Markdown files with the documentation text. Going deeper into technical details is beyond the scope of this paper, hence we provide an illustration of the whole generation process in figure 3.1. As you can see, we begin by designing a graph model in the Cinco product application, the WebDoc editor; clicking on the generate button starts a simultaneous creating of both the aforementioned project files. At this point, a fully functional documentation has already been created. We have chosen to generate the documentation file with references to placeholder pictures, that will be overridden once the Selenium script has been executed. To launch the documentation website two simple commands are needed. Then again, we will not go deeper in details to stay in the scope of our thesis, but there is a great online guide on how to get started with VuePress [31]. As for the Selenium-Java project, it can be imported in an integrated development environment (IDE) such as Eclipse and launch there. After successful execution, all the placeholder pictures will have been overridden and we have a complete documentation.

Chapter 4

Evaluation

The purpose of this chapter is to explain our approach and its results. First, we justify the decision for the chosen approach to the topic, as well as the challenges posed, and the solutions found for them. A brief description of the design pattern used, and the resulting benefits is also given. Subsequently, a minimal system is used to demonstrate the development of end user documentation. In addition, we present the resulting application, how it can be created and launched. Finally, the overall results and any alternative approaches will be discussed.

4.1 Method selection

Selenium is at the core of our methodology, as it is per se the default automated testing and manipulation framework for Web application. Not only the documentation of the API is thorough and exhaustive – since it is open-source – it also allows the manipulation of most of the browser engines through WebDrivers [26]. In addition to that, it integrates very well in a Java project as a Maven dependency, making it well-fitted for purposes. It must be noted that we do not intend to use Selenium for testing our application, rather than for automatically replicating the steps the documentation developer would have to take to create screen captures. In that sense, the Selenium WebDriver simulates the end user steps, captures the intermediate states of the Web application, and saves them as pictures in a dedicated folder.

Another state-of-the-art framework we chose is VuePress; it is well suited for creating static websites and is therefore perfect for designing technical documentation pages. Moreover, the effort required to configure a VuePress project is so low that the website can be up and running quickly. Alternatively, you can effortlessly bind the generated website to an existing domain, for example a company wiki page or similar. VuePress also has a great online documentation page, with a step-by-step guide on how to quickly setup a project and launch the server.

The link between the Selenium WebDriver and the VuePress project is the generated Java-Maven project. Here, we decided to relieve the documentation developer of the task of adjusting the project properties, adding the dependencies, naming the different packages and so on. This are the tasks an experienced java programmer would have to complete; and as we intend to empower non-programmer to be comfortable using our application, it is best we take care of the implementation of said application ourselves. We have also decided to generate dummy picture to make it possible to create the documentation with having the execute the Selenium script. This brings the advantage that different kind of picture can be added in place of the dummy ones in case the script is never intended to be executed.

Consequently, our editor application offered the capability to create a graphical model, which upon clicking the *generate button*, triggers the creation of both the Java application and the VuePress project structure, and subsequently, the Selenium WebDriver takes the indicated screenshots and saves them within the VuePress project folders.

4.2 Setup

The WebDoc editor application is a Cinco editor, which is based on Eclipse. Setting up the development environment is straight forward – at least if a certain acquaintance with the Eclipse environment already exists. Beforehand, a version of Java must be present on the system to run the application. Also, to automate a Web browser of choice, the corresponding Selenium WebDriver executable must be downloaded, and its location path must be added to the system's PATH variable. Concerning the Selenium libraries, we already take care of it by generating the pom.xml file, containing the required dependencies, along with the whole application structure.

After downloading and extracting the application package, it must be started just like a common Eclipse IDE would normally be. A splash screen presenting the application and indicating the progress of the launch process appears and within a few second a WebDoc IDE is started.

4.3 Results

Chapter 5

Final Remarks

5.1 Conclusion

We implemented a Web Application Documentor (WebDoc) that creates an end user documentation based on graph models using a graphical DSL. The use of graphical elements that resemble the actual HTML elements brings the advantages that the application is easy to use and building model graphs occurs almost intuitively. Also, we have added the implementation of module checks to assist the designer in the process of constructing valid model graph that will generate correct, executable application code.

Making use of the CINCO Meta Graph Language (MGL) and Meta Style Language (MSL), we determined the look of each model element of our graphical DSL by keeping a close fidelity to the actual HTML elements they represent. This allowed us to create an editor application, that enables the documentation designer to use those model elements to create graph diagram illustrating the needed user workflow. By applying model checking methods to validate resulting diagrams, we ensure that executable application code be generated within corresponding folder structure following established conventions.

We have been able to identify the characteristics of an end user documentation by following known standards and wisely apply the appropriate format, namely Markdown, to structure our documentation and choose VuePress as the right framework technology to transform it to a static website rendered in the Web browser.

The solution proposed in this thesis yields a well-structured and fully function documentation website. Our approach uses the Selenium WebDriver, which navigates the Web application in a short amount of time while taking screen captures at the indicated places. Writing such a Selenium script would not only require deep knowledge of the

Web automation framework but also good programming skills in Java. In our solution the WebDoc takes care of it for the developer.

The documentation developer would also benefit of the fact that the WebDoc conveniently creates the VuePress project with all the configuration files ready-to-go. This helps save a great amount of time, since a tutorial on how to create such a project is not necessarily needed.

5.2 Future Work

In the evaluation of the thesis, we have identified a couple of development opportunities for future improvements:

- **Cross-referencing model graphs**
  Reusability dictates that we be able to integrate complete model graphs in others to avoid modeling sequences multiple times. In this thesis, we achieved it by using the PrimeReference feature offered by the CINCO framework. Cross-referencing DocGraphModels inside other could be subject to improvements in the future, enabling us for example to list all available graph models in a separate view, where they would be visible to the WebDoc user and more intuitive to integrate to the currently edited model.

- **Implement more checks**
  We have implemented so far module checks that help build syntactically correct model graphs, by enforcing the constraints defined on the meta-specification level of our graphical DSL. In the future, we could implement more checks to assist the developer in validation some other aspects of the model. For example, we could validate that the string values for the Web element selectors have the correct syntax of CSS selectors or XPath selectors by validating them against a well-constructed regular expression.

- **Language extension**
  The example mentioned in the previous point raises the concern of allowing the developer to address Web elements by XPath and/or CSS. So far, we have implemented only the use of CSS selectors to address them. Many development frameworks for Web applications use dynamic CSS id attributes, which makes it challenging to use as selector for the WebDriver to find. This issue could as well be addressed in future version of our application.

One last point that requires our attention is the list of available Web elements to use while modeling. For those who are experienced in Web development, it is not surprising that new HTML elements be implement in the future. We fast-developing technologies and the growing number of devices that can launch browser application, we might have to adapt and offer more model elements to reflect new ones.

List of Figures

2.1 Development path of the end user documentation 6
2.2 Plain-text transformation with Markdown 8
2.3 Hierarchy of our graphical DSL specification 11
2.4 Impression of the TODO-App Web interface 17

3.1 Generation process of End user Documentation 19
3.2 Cinco Product Application - Graph model editor 20
3.4 Cinco product - Reusing Login graph model in the CreateNewList graph model 23
3.5 Example of a user workflow: here the login sequence 23
3.6 Login pane with screenshot of the highlighted input element 24
3.7 Structural mapping of models to Markdown 25
3.8 Project Validation view showing passed and failed checks 26
3.9 First approach: Sequential generation of the documentation 28
3.10 Second approach: Separate generation of the documentation 28

Listings

2.1 Excerpt from the feature.mgl, meta-specification of the FeatureGraphModel 12
2.2 Excerpt from the Doc.mgl, meta-specification of the DocGraphModel . . . 13
2.3 Excerpt from feature.style to be applied to feature.mgl 14
2.4 UserDocumentationTool.cpd 15

Glossary

DOM

The Document Object Model (DOM) is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. https://en.wikipedia.org/wiki/Document_Object_Model.

Ecore

The core Eclipse Modeling Framework (EMF) includes a metamodel (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.

Selenium

Selenium is a suite of tools for automating Web browsers. See https://www.selenium.dev/about/.

VuePress

VuePress is a Static Site Generator that generates pre-rendered static HTML for each page, and runs as an SPA once a page is loaded.

Abbreviations

API Application Programming Interface.

CPD Cinco Product Definition.

DIME DyWA Integrated Modeling Environment.

DOM Document Object Model.

DSL domain-specific language.

EMF Eclipse Modeling Framework.

HTML Hypertext Markup Language.

IDE integrated development environment.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Standards Organization.

jABC Java Application Building Center.

LOC lines of code.

MDD model-driven development.

MDSD model-driven software development.

MGL Meta Graph Language.

MSL Meta Style Language.

OOP Object Oriented Programming.

UI User Interface.

UML Unified Modeling Language.

WebDoc Web Application Documentor.

Bibliography

[1] *IEEE Standard for Adoption of ISO/IEC 26514:2008 Systems and Software Engineering–Requirements for Designers and Developers of User Documentation*, January 2011.

[2] *ISO/IEC/IEEE International Standard - Systems and software engineering – Requirements for managers of user documentation*, March 2012.

[3] *ISO/IEC/IEEE International Standard - Systems and software engineering - Requirements for managers of information for users of systems, software, and services*. ISO/IEC/IEEE 26511:2018(E), pages 1–90, 2018.

[4] *ISO/IEC/IEEE International Standard - Systems and software engineering–Software life cycle processes–Part 2: Relation and mapping between ISO/IEC/IEEE 12207:2017 and ISO/IEC 12207:2008*. ISO/IEC/IEEE 12207-2:2020(E), pages 1–278, 2020.

[5] *Definition of meta- prefix*. https://www.merriam-webster.com/dictionary/meta, November 2021. Accessed: November 02, 2021.

[6] Amalfitano, Domenico, Anna Rita Fasolino and Porfirio Tramontana: *Using dynamic analysis for generating end user documentation for Web 2.0 applications*. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 11–20. IEEE, September 2011.

[7] aside ufba: *GuidAutomator GitHub Repository*. https://github.com/aside-ufba/guide-automator. Accessed: November 04, 2021.

[8] Bettini, Lorenzo: *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[9] Bosselmann, Steve, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff and Bernhard Steffen: *DIME: A Programming-Less Modeling Environment for Web Applications*. In Margaria, Tiziana and Bernhard Steffen (editors): *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 809–832, Cham, 2016. Springer International Publishing.

[10] Brambilla, Marco, Jordi Cabot and Manuel Wimmer: *Model-driven software engineering in practice*. Synthesis lectures on software engineering, 3(1):1–207, 2017.

[11] Busch, Daniel and Annika Fuhge: *Cinco Product Specification*. https://gitlab.com/scce/cinco/-/wikis/Cinco-Product-Specification. Accessed: October 02, 2021.

[12] Cinco Dev. Team: *About Cinco SCCE Meta Tooling Framework*. https://cinco.scce.info/about/,. Accessed: July 14, 2021.

[13] Cone, Matt: *Markdown Guide | Getting Started*. https://www.markdownguide.org/getting-started/, October 2021. Accessed: November 02, 2021.

[14] Descher, Marco, Thomas Feilhauer and Lucia Amann: *Automated user documentation generation based on the Eclipse application model*. 04 2015.

[15] Despa, Mihai Liviu: *Comparative study on software development methodologies*. Database Systems Journal, 5(3):37–56, 2014.

[16] Eclipse Foundation: *Rich Client Platform*. https://wiki.eclipse.org/Rich_Client_Platform. Accessed: July 12, 2021.

[17] Eclipse.org: *Java with Spice*. https://www.eclipse.org/xtend/documentation/. Accessed: September 24, 2021.

[18] Fowler, Matin: *ModelDrivenSoftwareDevelopment*. https://martinfowler.com/bliki/ModelDrivenSoftwareDevelopment.html, 2008. Accessed: August 02, 2021.

[19] Kipyegen, Noela Jemutai and William PK Korir: *Importance of software documentation*. International Journal of Computer Science Issues (IJCSI), 10(5):223, 2013.

[20] MDN contributors: *HTML: HyperText Markup Language*. https://developer.mozilla.org/en-US/docs/Web/HTML. Accessed: November 02, 2021.

[21] Naujokat, Stefan: *Heavy meta: model-driven domain-specific generation of generative domain-specific modeling tools*. PhD thesis, Technical University of Dortmund, 2017.

[22] Naujokat, Stefan, Michael Lybecait, Dawid Kopetzki and Steffen Bernhard: *CINCO, a simplicity-drivern approach to full generation of domain-specific graphical modeling tools*. International Journal on Software Tools for Technology Transfer, 20(3), 2018.

[23] Oliveira, Allan dos Santos: *GuideAutomator: Automated user manual generation with Markdown*. 2016. Accessed: November 04, 2021.

[24] Pérez Andrés, Francisco, Juan de Lara and Esther Guerra: *Domain Specific Languages with Graphical and Textual Views*. In Schürr, Andy, Manfred Nagl and Albert Zündorf (editors): *Applications of Graph Transformations with Industrial Relevance*, pages 82–97, Berlin, Heidelberg, October 2008. Springer Berlin Heidelberg.

[25] SCCE Group: *Sustainable Computing for Continuous Engineering*. https://www.scce.info/. Accessed: September 29, 2021.

[26] Software Freedom Conservancy: *Getting Started | Selenium*. https://www.selenium.dev/documentation/getting_started/, October 2021. Accessed: September 20, 2021.

[27] Stahl, Thomas, Markus Voelter and Krzysztof Czarnecki: *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., Hoboken, NJ, USA, July 2006.

[28] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges and Christian Kubczak: *Model-Driven Development with the jABC*. In Bin, Eyal, Avi Ziv and Shmuel Ur (editors): *Hardware and Software, Verification and Testing*, pages 92–108, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[29] TU Dortmund, Chair of Programming Systems: *Java Application Building Center*. http://ls5-www.cs.tu-dortmund.de/projects/jabc/index.php. Accessed: October 04, 2021.

[30] Waits, Todd and Joseph Yankel: *Continuous system and user documentation integration*. In *2014 IEEE International Professional Communication Conference (IPCC)*, pages 1–5, 2014.

[31] You, Evan: *VuePress | Vue-powered Static Site Generator*. https://vuepress.vuejs.org/, October 2021. Accessed: November 02, 2021.

I hereby certify that I have written this paper independently and have not used any sources or aids other than those indicated, and that I have clearly marked any citations.

Dortmund, November 9, 2021

Mukendi Mputu