Defining Domain Language of Graphical User Interfaces

Michaela Bačíková, Jaroslav Porubän, and Dominik Lakatoš

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
{michaela.bacikova, jaroslav.poruban, dominik.lakatos}@tuke.sk

Abstract

Domain-specific languages are computer (programming, modeling, specification) languages devoted to solving problems in a specific domain. The least examined DSL development phases are analysis and design. Various formal methodologies exist, however domain analysis is still done informally most of the time. There are also methodologies of deriving DSLs from existing ontologies but the presumption is to have an ontology for the specific domain. We propose a solution of a user interface driven domain analysis and we focus on how it can be incorporated into the DSL design phase. We will present the preliminary results of the DEAL prototype, which can be used to transform GUIs to DSL grammars incorporating concepts from a domain and thus to help in the preliminary phases of the DSL design.

1998 ACM Subject Classification D.2.11 Software Architectures, D.2.12 Interoperability, D.2.13 Reusable Software, H.1.2 User/Machine Systems, H.5.2 User Interfaces (D.2.2, H.1.2, I.3.6)

Keywords and phrases graphical user interfaces, domain analysis, formalization, domain-specific languages, DEAL

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.187

1 Introduction

Programming languages are used for human-computer interaction. Domain-specific languages (DSLs) such as Latex, SQL, BNF, are computer languages tailored to a specific application domain [15, 30, 48, 41, 21, 19]. In contrast, general-purpose languages (GPLs) such as Java, C and C# are designed to solve problems in any problem area.

When developing a new software, a decision must be made as to which type of programming language will be used: GPL or DSL. The issue is further complicated if an appropriate DSL does not exist. The reasons for using DSLs instead of GPLs are: easier programming, reuse of semantics, easier verification and understandability (and programmability) for end-users [15, 30]. However, the cost of developing a new DSL is usually high [15, 22] because it involves development of language parsers and generators along with the language.

A DSL should be developed whenever it is necessary to solve a problem that belongs to a problem family and when we expect more problems from the same family of problems to appear in the future. The implementation phase is well documented by many researchers [21] but the analysis and design phases are still dropped behind. The various DSL development phases and the tool support of each of them is very nicely described in the article by Čeh et al [10].

On the other hand, various methodologies for domain analysis (listed in the related work section) were developed such as DSSA, FODA, ODM. But they are often not used due to their complexity and the DA is done informally. This complicates the development od DSLs.

2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 187–202

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Even if a formal methodology is used for performing DA there are no clear guidelines on how the output from DA can be used in a language design process. Guillermo et al. [4] identified the following types of DA outputs:

a) *a domain model describing the target domain* - contains domain dictionary, terms, concepts, their relations and properties,
b) *a domain model describing how to develop systems* in the target domain.

The concrete implementations of the DA outputs vary when referring both to the first or the second type specifically. We identified the following DA output types:

- *simple domain dictionary* - contains terms, hierarchy of terms,
- *lexicographical dictionary* - terms, lexicographical relations (such as hyponymy, synonymy, etc.),
- *ontology* - terms, hierarchy, relations (such as is-a, part-of, has-part, regulates, exclusive-to relations, etc.),
- *feature diagram* - focuses on feature commonalities and variabilities in product lines, contains relations between the features representing the cases whether they are used in the product or not (and, or and alternative),
- *design documentation* - describing the system model or meta-model (class diagram, entity-relationship model, data-flow diagram, state-transition model, etc.),
- *reusable software artifacts* - fragments of code, libraries, reusable frameworks, etc.,
- *EBNF* - a grammar defining the domain language syntax.

Not each of these outputs can be used as an input to the DSL design phase. As proposed by Čeh et al. [10], it is possible to use ontologies. Their paper outlines the method of transformation of ontologies into DSL grammars and they have implemented the Ontology2DSL framework, which is able to demonstrate this transformation. By this they try to contribute to the first phases of DSL development, analysis and design.

However, the assumption for their solution is the existence of an ontology designed for the given specific domain. Which is actually an equally difficult problem compared to finding an existing DSL for the given specific domain.

In this paper we focus on a similar problem, but in the area of existing software systems. Many formal methods exist for analyzing the existing software systems with the goal of performing domain analysis (examples of them are listed and described in the section 6). The results in many cases represent a model or a design of a new software system (such as the class diagram); or a library of reusable software artifacts; both hardly usable for the DSL analysis or design.

The reason is that it is very difficult to filter out the implementation details from the target application so only the relevant domain information would remain. Therefore instead of trying to extract domain information, the existing methods rather benefit from the system's implementation details by extracting reusable code fragments or libraries and software documentation.

Thus the existing approaches for extracting domain-relevant information in a form of domain descriptions focus rather on existing informal documents and domain experts as sources of domain knowledge. The main drawback of such resources is their non-formal nature. Documents are mainly processed by complicated techniques such as Natural Language Processing (NLP) and it is necessary to verify the results manually. On the other hand, the information from domain experts has to be gained by personally meeting with them because often-times they are not willing to (nor they are capable of) writing the information in a

semi-formal manner by themselves. The consecutive process of formalization of the gained data is even more tedious.

In our approach we do not focus on the extraction of domain information from the existing software systems in general. Our main points of interest are graphical user interfaces (GUIs) made of components. In the scope of supporting the early DSL development phases, our assumption is the existence of a user interface for the given specific domain, which nowadays is nothing unusual. Often-times there is a need to create a new software system and the obsolete version is thrown away without any regards to its reuse possibilities. There is almost an endless number of component-based applications for different domains and even a bigger number of web applications.

But the amount of existing software systems is *certainly larger* than the amount of existing ontologies. Therefore we claim that our approach is more efficient than the approach of Čeh et al. However the benefit resulting from their approach - no need to start from scratch but with a generated DSL grammar - is preserved in our approach.

In this paper we propose a formal design of a methodology for deriving DSL grammars from existing user interfaces. We also present the DEAL method for traversing GUIs and the preliminary results of the DEAL prototype, which was designed for the purpose of domain information extraction from GUIs.

The research thesis for this paper stands as follows: If an application exists for a specific domain with a GUI made of components and we have reflection available along with the possibility of identifying the component structure, then it is possible to design a tool, which uses reflection, and which can traverse the application GUI and create a DSL design from it. This DSL design can be then edited by a language designer.

The presumption for using DSLs is an existence of a family of a repetitive problem. If the repetitive problem is a creation of a new DSL, creation of a new GUI or writing/performing commands in GUIs, then this paper tries to propose a partial solution for this problem.

The organization of this paper is as follows. Section 2 is intended to demonstrate an example of a GUI to DSL transformation. Section 3 presents the transformation rules used for generating a a DSL from a GUI and from different GUI components. Sections 4 and 5 present the DEAL method for traversing GUI components and the DEAL tool prototype which is an implementation of the DEAL method. The work related to this paper and the conclusion are summarized in Section 6.

2 GUI is a Language Definition

What do we see when we look at any user interface? Windows, dialogs, buttons, menus, labels, textfields, components. However a GUI is more than that, it is a definition of a DSL. We will try to demonstrate this fact on a simple example.

Let us look at fig. 1 with a form containing information about a person. Its concrete syntax could be specified using a grammar, where the non-terminals are noted by first capital letter and the terminals are noted in ⟨ ⟩ brackets. The elements ⟨STRING⟩ and ⟨NUMBER⟩ represent a terminal string or numeric value. The concrete syntax of the domain language

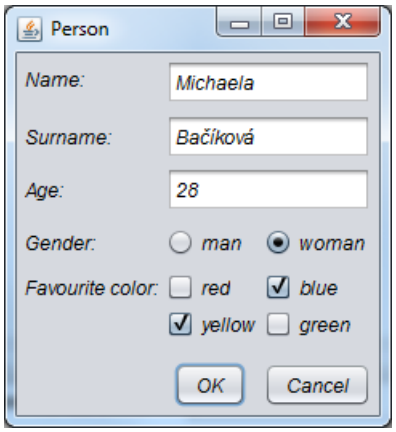


Figure 1 An example of a person form.

for the person form can be defined as follows:

$$
\begin{aligned}
\mathit{Person} &\rightarrow \langle\text{Person}\rangle \mathit{Name\ Surname\ Age\ Gender\ FavouriteColor} \\
\mathit{Name} &\rightarrow \langle\text{Name}\rangle\ \langle\text{STRING}\rangle \\
\mathit{Surname} &\rightarrow \langle\text{Surname}\rangle\ \langle\text{STRING}\rangle \\
\mathit{Age} &\rightarrow \langle\text{Age}\rangle\ \langle\text{STRING}\rangle \\
\mathit{Gender} &\rightarrow \langle\text{Gender}\rangle\ \langle\text{man}\rangle \mid \langle\text{woman}\rangle \\
\mathit{FavouriteColor} &\rightarrow \langle\text{Favourite colors}\rangle\ (\ \langle\text{red}\rangle?\ \langle\text{blue}\rangle?\ \langle\text{green}\rangle?\ \langle\text{yellow}\rangle?\ )
\end{aligned}
$$

If the *age* text field was a formatted text field, or a spinner (fig. 2 on the right), we could extract the input type: string, number or date. The grammar rule for age could then look as follows:

$$\mathit{Age} \rightarrow \langle\text{Age}\rangle\ \langle\text{NUMBER}\rangle$$

We showed that the GUI defines a language. What is the sentence in this language? And where is the semantics?

A sentence in this language could look as follows:

> *Person Name Michaela Surname Bacikova Age* 28 *Gender woman*
> *Favourite colors* (*blue yellow*)

And writing such sentences in the GUI language means filling the form with values.

The second question is the semantics, which is defined by the application. The semantic meaning of clicking the *OK* button is defined in the code. So it holds for every single component contained in a UI.

In the next section we will describe how the GUI language specification can be automatically created from an existing user interface.

Figure 2 The *age* input field represented by a spinner with a numeric value.

3 Method for DSL Specification Derivation

For a better explanation of the process of the domain language specification derivation, we present a sketch of transformation for basic Java components. This is to be a formal description of our extraction method called *DEAL*. The transformation is defined as a semantic function, which expresses the domain meaning with the goal of creating domain grammar. The transformation function is defined as follows:

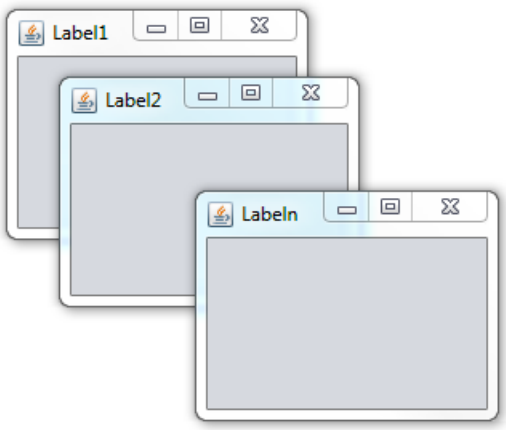
$$\mathcal{T} : Component \times ComponentTree \longrightarrow Production$$

where $\mathcal{T}$ is the transformation function, *Component* is the semantic domain of components and *Production* is the semantic domain of EBNF production rules. In order to extract any information from components we also have to know their location in the tree, to get the information about the component's parents and child components. Therefore the input of the semantic function also involves the *ComponentTree* semantic domain of component hierarchies in a form of trees. For example, in tabbed pane the tabs are located in the tabbed pane and their tab names (labels) are stored in the tabbed pane. Therefore we need to know the information from the parent component and store it to the child component.

Here we will present the transformation rules. Each of the rules is always labelled with the component name above the left semantic bracket.

We assume that the processing always starts with a GUI of an application which is component-based. Each UI should consist of one or more *scenes*[1]. Opening of a new scene can be initialized by starting the application or by performing action on a functional component.

$$\mathcal{T}_{GUI}\left[\!\!\left[\right.\right.$$

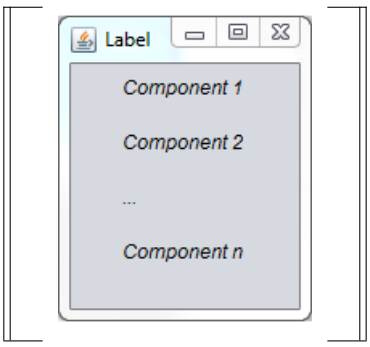


$$\left.\left.\right]\!\!\right] = GUI \rightarrow Label_1 \; Label_2 \ldots Label_n$$

Each scene can be identified by its identifier (if there is any), e.g. a window title, a dialog title, a web page title. This holds for each component: it is identified by its name or description. We note the identifier of a scene or of a component generally as "*Label*" and it represents a string of characters. Each scene can contain a graphical content - a list of components.

[1] As defined by Kösters in [23].

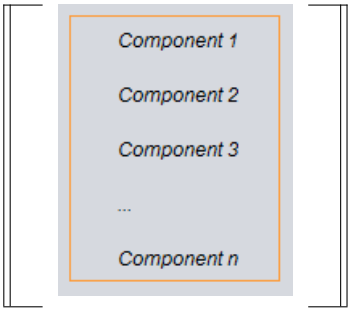
Scene

$\mathcal{T}$ 

$= \quad Label \rightarrow \langle \text{Label} \rangle \; Component_1 \; Component_2 \; \ldots \; Component_n$

The components can be logically arranged in an unlimited number of containers. Containers are components which can contain other components (e.g. panel or a tabbed pane).

Panel

$\mathcal{T}$ 

$= \quad Panel \rightarrow Component_1 \; Component_2 \; \ldots \; Component_n$

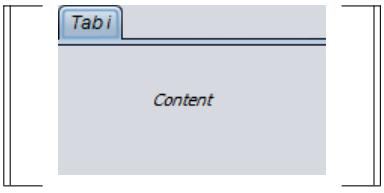
From a tabbed pane, additional information can be extracted. Since the containers are stored in tabs and these tabs have a tab name (Tab_i), the tab name can be extracted as an identifier of the group of components contained in the given panel.

Tabbed pane

$\mathcal{T}$ 

$= \quad TabbedPane \rightarrow Tab_1 \; Tab_2 \; \ldots \; Tab_n$

Tab

$\mathcal{T}$ 

$= \quad Tab \rightarrow \langle \text{Tab} \rangle \; Content$

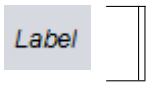
Textfields are components which usually have no label integrated in their implementation. However the label can be (optionally) connected to them in the implementation phase by using the *for* attribute in the label component (the for attribute is supported e.g. by the Java language and by HTML) where the reference to the described component is inserted. This holds for any other type of component. Moreover, if the textfield is a formatted textfield, the type of the input value (string, numeric, date, etc.) and restrictions on the input text can be extracted (such as regular expression or min/max value).

Textfield

$\mathcal{T}$ [] $=$ *Label* $\rightarrow$ ⟨Label⟩ ⟨STRING⟩

We realize that setting the label *for* attribute is an optional issue, however if the programmer does not set it, we perceive it as a usability issue. Once the label was assigned to the *for* component and saved, it has no meaning in the result of the domain analysis process therefore it can be thrown away. The important issue is the component the label describes.

Label

$\mathcal{T}$ [] $=$ *Label* $\rightarrow \epsilon$

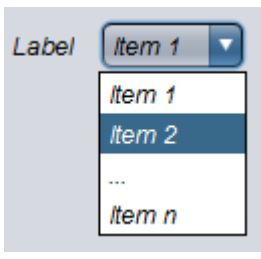
The functional components (such as buttons, menu items, web links, etc.) have a similar rule. The reason for excluding functional components is that the labels are not important for the domain. The semantics of functional components is the execution of the action in GUI, which is defined by them and stored in the code. We see the execution of events in the user interface as a completely different chapter, which belongs rather to the areas of usability and automated GUI testing.

Button

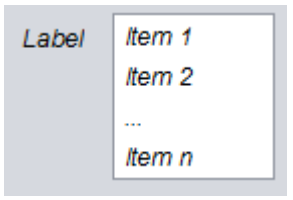
$\mathcal{T}$ [] $=$ *Label* $\rightarrow \epsilon$

The comboboxes, lists and checkbox and radio button lists are examples of components (or lists of components) representing *lists of terms*. Depending on whether the component (or a group of components) has a single selection or multiple selection set, the relations between the terms are derived. Mutual exclusive relation (single selection) is represented by alternatives, terms that are not mutually exclusive (multiple selection) are represented by one or zero occurence ($?$) of each term.

Combobox

$\mathcal{T}$ [] $=$ *Label* $\rightarrow$ ⟨Label⟩ (⟨Item$_1$⟩ | ⟨Item$_2$⟩ | ... | ⟨Item$_n$⟩)

List

$\mathcal{T}$ [] $=$ *Label* $\rightarrow$ ⟨Label⟩ (⟨Item$_1$⟩ | ⟨Item$_2$⟩ | ... | ⟨Item$_n$⟩)

Checkbox list

$$\mathcal{T}\left[\begin{array}{l}\square\ \textit{Item 1}\\ \blacksquare\ \textit{Item 2}\\ \quad\ldots\\ \blacksquare\ \textit{Item n}\end{array}\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle\ \ \langle\text{Item}_1\rangle\ ?\ \ \langle\text{Item}_2\rangle\ ?\ \ldots\ \ \langle\text{Item}_n\rangle\ ?$$

Radio button list

$$\mathcal{T}\left[\begin{array}{l}\bigcirc\ \textit{Item 1}\\ \bullet\ \textit{Item 2}\\ \quad\ldots\\ \bigcirc\ \textit{Item n}\end{array}\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle\ \ (\ \langle\text{Item}_1\rangle\ \mid\ \langle\text{Item}_2\rangle\ \mid \ldots \mid\ \langle\text{Item}_n\rangle\ )$$

In the examples the list was initialized with single selection. If it would have been initialized with multiple selection the rule would be the same as for the checkbox list.

Spinners (like formatted textfields) in the Java language are instantiated with a model defining the restrictions on the data which are stored in them. The spinner model can be chosen of three basic types: List, Number and Date. Each of the model types has its particular type set and if it is a Number type, the data type of the number can be extracted (Float, Double, Integer, Binary, etc.).

Spinner

$$\mathcal{T}\left[\textit{Labe}\ \boxed{0}\ \right] = \textit{Label} \rightarrow \langle\text{Label}\rangle\ \ \langle\text{NUMBER}\rangle$$

If the spinner has default values of a list of items, then it has the same rule as any component with single selection (alternatives). In the model, the minimum, maximum and default values are also set and this information can be extracted as additional information about the term.

Date spinner

$$\mathcal{T}\left[\textit{Label}\ \boxed{5/13/13\ 1:13\ PM}\ \right] = \textit{Label} \rightarrow \langle\text{Label}\rangle\ \ \langle\text{DATE}\rangle$$

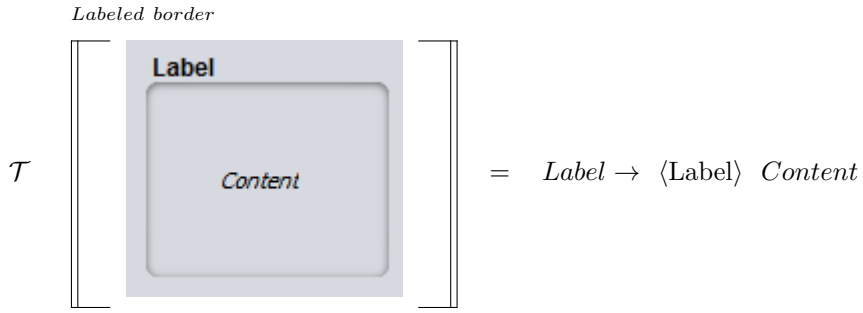
Sliders are similar to spinners, with the difference of displaying the chosen value not by textual representation, but on a graphical scale. Maximum and minimum values can be extracted.

Slider

$$\mathcal{T}\left[\begin{array}{l}\textit{Label}\\ \textit{min}\qquad\textit{max}\end{array}\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle\ \ \langle\text{NUMBER}_{\text{min-max}}\rangle$$

In many types of containers (such as panels) the label (or title) attribute is optional. Therefore if a group of components (representing terms) is logically related, it is not clear what exactly they have in common, whether they belong to the same subdomain. However some containers have identifiers which give a clue about the subdomain of their content. The example of such a container is a panel with the labeled border. Each scene should have a

label (window title, dialog title, webpage title, etc.) and this name gives a clue about the application domain or subdomain. Each component contained in the scene belongs to this domain. This way the hierarchy of terms is created.



According to the illustrated rules we can see that on various component types apply various extraction conditions and it is possible to derive basic relations between the terms and properties of terms such as default value, minimum and maximum value and restrictions. A hierarchy of domains and subdomains (a part-of or a belongs-to relation) is created by containers placed in other containers and in scenes. The result of the extraction method is a formalized domain model. The $\langle$STRING$\rangle$, $\langle$NUMBER$\rangle$ and $\langle$DATE$\rangle$ could be defined as follows:

$\langle \text{STRING} \rangle \rightarrow *$
$\langle \text{NUMBER} \rangle \rightarrow [0-9]+$
$\langle \text{DATE} \rangle \rightarrow [0-9][0-9]/[0-9][0-9]/[0-9][0-9]\ ([0-9][0-9]{:}[0-9][0-9]\ (\text{AM}|\text{PM}))?$

4 The DEAL Method

We have performed a research in the area of *automated domain analysis of user interfaces*, described in detail in [24] [8] and [6]. Our general goal was to prepare the first phase for automatized analysis of UI domain content by means of automatic extraction of domain content (terms), the properties of the concepts and structure and analysis of relations between the content units. The method of traversing GUIs and extraction of domain information from existing GUIs is called DEAL (Domain Extraction ALgorithm). The input of DEAL is an existing system programmed in a language, which provides the possibility of determining the component structure, reflection and/or aspect-oriented programming. The output of DEAL is a domain model displayed in fig. 3.

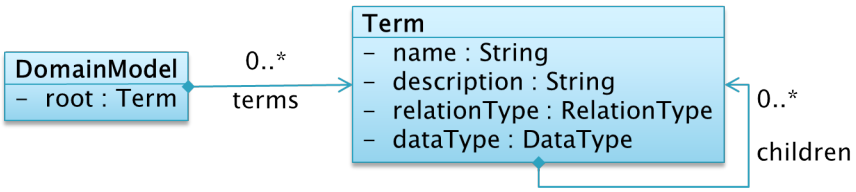


Figure 3 Domain model representation in DEAL.

The DEAL traversal algorithm was described in [6] and in the section 3 we described the method of transformation.

The domain content of the target UI is extracted as a graph of terms, their relations and properties. Some relations (such as the parent-son relation) are explicit, other relations (such as mutual exclusivity) are derived automatically based on the typology of components

and, partially, based on their topology as outlined in this paper. From this domain model representation, different types of domain models, including grammars and ontologies, can be generated. We experimentally confirmed the possibility of extraction of a graph of domain terms including the derivation of relations between some terms on applications in Java language and we performed a feasibility analysis of web application analysis (HTML) [37, 6, 7]. Both languages meet the presumption to have the component nature, they provide the possibility to determine the component structure.

5 The DEAL Tool Prototype

The DEAL tool prototype[2] is a software solution for extracting domain content of existing user interfaces and it proves the possibility of using the DEAL method on Java applications. Currently, DEAL uses YAJCo[3] language processor to generate grammars. More about YAJCo can be found in [38].

The DEAL tool prototype proves that it is possible to:

- traverse the GUI of an application, which is made of components,
- extract domain information from an existing GUI in a formalized form,
- and to generate a DSL grammar based on the extracted information.

The DEAL prototype was tested on 17 open-source Java applications, all of which are included in the DEAL project online. It is however still in development and we are improving it based on the test results.

Some Java applications have their own class loaders, therefore we had to use AspectJ to be able to extract information from them. The tutorial, documentation and related references are all published online along with the tool[4].

An example of the DEAL output for the person form (fig. 1) is the grammar generated from the extracted model:

$$
\begin{aligned}
\mathit{Person} &::= (\langle\text{Person}\rangle \mathit{Name}\ \mathit{Surname}\ \mathit{Age}\ \mathit{Gender}\ (\mathit{Favourite\_color})*) \\
\mathit{Name} &::= (\langle\text{Name}\rangle\ \langle\text{STRING\_VALUE}\rangle) \\
\mathit{Surname} &::= (\langle\text{Surname}\rangle\ \langle\text{STRING\_VALUE}\rangle) \\
\mathit{Age} &::= \langle\text{Age}\rangle\ \langle\text{STRING\_VALUE}\rangle \\
\mathit{Gender} &::= \langle\text{Gender}\rangle\ (\langle\text{man}\rangle \mid \langle\text{woman}\rangle) \\
\mathit{Favourite\_color} &::= \langle\text{Favourite color}\rangle\ (\langle\text{red}\rangle\ \langle\text{blue}\rangle\ \langle\text{green}\rangle\ \langle\text{yellow}\rangle\ )
\end{aligned}
$$

The grammar is in YAJCo notation and the rules for not-mutually exclusive terms are supplemented by the $0-n$ version because YAJCo does not support the ? operator. However the results show that generating DSL grammars from existing user interfaces is definitely possible.

[2] DEAL project can be found at: https://www.assembla.com/spaces/DEALtool
[3] YAJCo project can be found at: https://code.google.com/p/yajco/ and YAJCo maven project at: http://mvnrepository.com/artifact/sk.tuke.yajco
[4] DEAL project documentation and tutorials: https://www.assembla.com/spaces/DEALtool/wiki

6 Related Work

Here we briefly summarize the different approaches related to: domain analysis, ontology extraction, GUI modeling and semantic UIs and reverse engineering.

Domain Analysis The domain analysis was first defined by Neighbors [33] in 1980 and he stresses that domain analysis is the key factor for supporting reusability of analysis and design, not the code.

The most widely used approach for domain analysis is the *FODA* (Feature Oriented Domain Analysis) approach [17]. FODA aims at the analysis of software product lines by comparing the different and similar features or functionalities. The method is illustrated by a domain analysis of window management systems and explains what the outputs of domain analysis are but remains vague about the process of obtaining them. Very similar to the FODA approach, and practically based on it, is the *DREAM* (Domain Requirements Asset Manager) approach by Mikeyong et. al. [31]. They perform commonality and variability analysis of product lines too, but with the difference of using an analysis of domain requirements, not features or functionalities of systems. Many approaches and tools support the FODA method, for example Ami Eddi [12], CaptainFeature [1], RequiLine [2] or ASADAL [25]. Other examples of formal methodologies are ODM (Organization Domain Modeling) [45] and DSSA (Domain Specific Software Architectures) [47].

There are also approaches that do not only support the process of domain analysis, but also the reusability feature by providing a library of reusable components, frameworks or libraries. Such approaches are for example the early *Prieto-Díaz approach* [13] that uses a set of libraries; or the later *Sherlock* environment by Valerio et. al. [54] that uses a library of frameworks.

The latest efforts are in the area of *MDD* (Model Driven Development). The aim of MDD is to shield the complexity of the implementation phase by domain modelling and generative processes. The MDD principle support provides for example the Czarnecki project Clafer [5] and the FeatureIDE plug-in [50] by Thüm and Kästner.

ToolDay (A Tool for Domain Analysis) [27] is a tool that aims at supporting all the phases of domain analysis process. It has possibilities for validation of every phase and a possibility to generate models and exporting to different formats.

All these tools and methodologies support the domain analysis process by analysing data, summarizing, clustering of data, or modelling features. But the input data for domain analysis (i.e. the information about the domain) always comes from the users, or it is not specified where it actually comes from. Only the *DARE* (Domain analysis and reuse environment) tool from Prieto-Díaz [16] primarily aims at automatic collection and structuring of information and creating a reusable library. The data is collected not only from human resources, but also *automatically from existing source codes and text documents*. But as mentioned above, the source codes do not have to contain the domain terms and domain processes. The DARE tool *does not analyse the GUIs* specifically.

Last but not least, the approach most similar to ours is the one proposed by Čeh et al. [10]. They proposed a methodology of transforming existing ontologies into DSL grammars and they present the results of their Ontology2DSL framework. The disadvantage in comparing to our approach is the little amount of existing ontologies available when comparing to the amount of existing software systems.

Ontology Extraction Many approaches are targeted to ontology learning. Several methodologies for building ontologies exist, such as OTK, METHONTOLOGY or DILIGENT, but they target ontology engineers and not machines [9]. Many methods and different sources of analysis are used to generate ontologies automatically. Among the Results are almost always combined with a manual controlling and completing by a human and as an additional input, almost always some general ontology is present (a "core ontology") serving as a "guideline" for creating new ontologies. Different methods are used to generate ontologies:

1. clustering of terms [44, 58, 39],
2. pattern matching [51, 58, 56, 39],
3. heuristic rules [51, 56, 39],
4. machine learning [34, 43],
5. neural networks, web agents, visualizations [39],
6. transformations from obsolete schemes [56],
7. merging or segmentation of existing ontologies [42, 58],
8. using fuzzy logic to generate a fuzzy ontology, which can deal with vague terms such as FFCA method ([40] and [11]) or FOGA method [49],
9. analysis of web table structures [35, 51, 26],
10. analysis of fragments of websites [57].

A condition for creating a good ontology is to use many sources as an input to analysis - structured, non-structured or semi-structured - and to use a combination of many methods [9]. Therefore as an additional mechanism for identifying different types of relationships (e.g. mutual exclusivity, hierarchical relations), WordNet web dictionary [35, 3, 14, 58] or other web dictionaries or databases available on the Web are used. A state of the art from 2007 can be found in [9].

GUI Modelling and Semantic User Interfaces Special models are designed specifically for modelling UIs or for modelling the interaction with UIs, whether they are older, such as CLASSIC language by Melody and Rugaber [32], or modern languages, such as XML, described in the review made by Suchon and Vanderdonckt in [46]. Paulheim [36] designed UI models of interaction with users. For UI configurations usually models such as configuration ontology designed for WebProtégé tool in [53] are used.

The most complex UI model was designed by Kösters in [23] as a part of the modelling process of the FLUID method for combined analysing of UIs and user requirements. A part of Kösters model is a domain model and model of UI (UIA-Model). Our model was slightly inspired by Kösters work - however we use domain-specific modelling without the relation to user requirements, therefore our model is simpler.

An interesting work was made in the area of semantic UIs by Porkoláb in [52].

Reverse Engineering Only a few works in the area of reverse engineering will be mentioned since our work is primarily focused on the area of domain analysis, not on reverse engineering. However, there are several works closely related to our work. Specifically, they are either reverse processes compared to ours (i.e. generate GUIs from domain models), or they produce other outputs than a DSL grammar:

- a GUI-driven generating of applications by Luković et al. in [29],
- generating of UIs based on models and ontologies by Kelshchev and Gribova in [18],
- deriving UIs from ontologies and declarative model specifications by Liu et al. in [28],
- program analysing and language inference [20].

A very interesting process is also seen in [55] where authors transform ontology axioms into application domain rules however the results are not as formal as our DSL grammar.

7 Conclusion

In this paper we presented the actual state of our research and introduced our DEAL method for extracting domain information from GUIs. We showed that the utilization of this method is not strictly in the area of the domain analysis, with the intent of creating a new software system. We argue that it is possible to define a domain-specific language. A GUI is a template that defines how the input accepted by the GUI should look like. According to this template it is possible to extract (generate) a GUI domain language specification, which, analogically, represents a *formalized* template for the GUI inputs – according to it, it is possible to determine whether a sentence belongs to the domain language defined by the GUI, or not.

The generated specification of the domain language defined by the UI can be utilized in further processes, such as automatic filling of forms. It happens in real cases, such as organizing conferences (or other events, where people need to register) where many people submit papers. The conference organizers get the information from a submitting system including the submitter names, surnames, personal information and information about their submitted papers. All this data have to be entered into the department software system and since there is no automated transfer system, they have to be entered manually. For each single submitter the whole form has to be filled manually again and again. And if there are 1000 submitters, we can imagine that it takes a lot of effort and time.

Using our approach, the conference organizer would automatically derive a grammar for the given language based on the GUI, and using this grammar it is possible to create a tool for automatic form filling. Moreover, this tool can be generated based on the grammar specification and it would also check the input for correctness.

Acknowledgements This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution. Thanks to Milan Nosáľ for brainstorming and reviewing the paper.

References

1 Captainfeature, the webpage of captainfeature sourceforge.net project. https://sourceforge.net/projects/captainfeature, 2005. [Online 2013].

2 The webpage of requiline project. http://wwwlufgi3.informatik.rwth-aachen.de/TOOLS/requiline/, 2005. [Online 2013].

3 Y. J. An, J. Geller, Y. Wu, and S. A. Chun. Automatic generation of ontology from the deep web. In *Database and Expert Systems Applications, 2007. DEXA '07. 18th International Workshop on*, pages 470–474, Sept.

4 G. Arango. A brief introduction to domain analysis. In *Proceedings of the 1994 ACM symposium on Applied computing*, SAC '94, pages 42–46, New York, NY, USA, 1994. ACM.

5 K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag.

6 M. Bačíková and J. Porubän. Analyzing stereotypes of creating graphical user interfaces. *Central European Journal of Computer Science*, 2:300–315, 2012.

7 M. Bačíková, J. Porubän, and Lakatoš D. Introduction to domain analysis of web user interfaces. In *Proceedings of the Eleventh International Conference on Informatics, INFORMATICS'2011*, pages 115–120, Rožňava, Slovakia, 2011.

8 Michaela Bačíková. Domain analysis with reverse-engineering for gui feature models. In *POSTER 2012 : 16th International Student Conference on Electrical Engineering*, volume 16, pages 1–5. Czech Technical University in Prague, May 2012.

9 I. Bedini and B. Nguyen. Automatic ontology generation: State of the art. Technical report, University of Versailles, December 2007.

10 I. Čeh, M. Crepinsek, T. Kosar, and M. Mernik. Ontology driven development of domain-specific languages. *Computer Science and Information Systems*, (2):317–342, 2011.

11 W. Chen, Q. Yang, L. Zhu, and B. Wen. Research on automatic fuzzy ontology generation from fuzzy context. In *Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation - Volume 02*, ICICTA '09, pages 764–767, Washington, DC, USA, 2009. IEEE Computer Society.

12 K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 156–172, London, UK, 2002. Springer-Verlag.

13 R. P. Díaz. Reuse Library Process Model. Final Report. Technical report start reuse library program, Electronic Systems Division, Air Force Command, USAF, Hanscomb AFB, MA, 1991.

14 Y. Ding, D. Lonsdale, D. W. Embley, and Li Xu. Generating ontologies via language components and ontology reuse. In *In Proceedings of 12th International Conference on Applications of Natural Language to Information Systems (NLDB'07*, 2007.

15 M. Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, October 2010.

16 W. Frakes, R. Prieto-Diaz, and Ch. Fox. Dare: Domain analysis and reuse environment. *Ann. Softw. Eng.*, 5:125–141, January 1998.

17 K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

18 A. Kelshchev and V. Gribova. From an ontology-oriented approach conception to user interface development. In *ITHEA, International Journal ITA (Institue of Mathematics and Informatics, Bulgarian Academy of Sciences)*, volume 10. Institute of Information Theories and Applications FOI ITHEA, 2003.

19 J. Kollár and S. Chodarev. Extensible approach to dsl development. *Journal of Information, Control and Management Systems*, 8(3):207–215, 2010.

20 J. Kollár, S. Chodarev, E. Pietriková, L. Wassermann, D. Hrnčič, and M. Mernik. Reverse language engineering: Program analysis and language inference. In *Informatics'2011 : proceedings of the Eleventh International Conference on Informatics*, pages 109–114. Košice, TU, November 16-18 2011.

21 T. Kosar, P. E. M. López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.

22 T. Kosar, N. Oliveira, M. Mernik, M. J. V. Pereira, M. Črepinšek, D. da Cruz, and P. R. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

23 G. Kösters, H. W. Six, and J. Voss. Combined analysis of user interface and domain requirements. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, ICRE '96, pages 199–, Washington, DC, USA, 1996. IEEE Computer Society.

24 Michaela Kreutzová (Bačíková), Jaroslav Porubän, and Peter Václavík. First step for gui domain analysis: Formalization. *Journal of Computer Science and Control Systems*, 4(1):65–70, 2011.

25 Postech Software Engineering Laboratory. A review of asadal case tool.

26 X. Lei and R. Yong. Ontology generation from web tables: A 1+1+n approach. In *Proceedings of the 2010 International Forum on Information Technology and Applications - Volume 01*, IFITA '10, pages 234–239, Washington, DC, USA, 2010. IEEE Computer Society.

27 L. Lisboa, V. Garcia, E. de Almeida, and S. Meira. Toolday: a tool for domain analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 13:337–353, 2011.

28 B. Liu, H. Chen, and W. He. Deriving user interface from ontologies: A model-based approach. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '05, pages 254–259, Washington, DC, USA, 2005. IEEE Computer Society.

29 I. Luković, S. Ristić, A Popović, and P. Mogin. An approach to the platform independent specification of a business application. In *Central European Conference on Information and Intelligent Systems*, 2011.

30 M. Mernik, J. Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

31 M. Moon, K. Yeom, and H. Seok Chae. An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *IEEE Trans. Softw. Eng.*, 31:551–569, July 2005.

32 M. M. Moore and S. Rugaber. Domain analysis for transformational reuse. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.

33 J.M. Neighbors. *Software construction using components*. PhD thesis, University of California, Irvine, 1980.

34 B. Omelayenko. Learning of ontologies for the web: the analysis of existent approaches. In *In Proceedings of the International Workshop on Web Dynamics*, 2001.

35 Aleksander P. Automatic ontology generation from web tabular structures. *AI Communications*, 19:2006, 2005.

36 H. Paulheim. Ontologies for user interface integration. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 973–981, Berlin, Heidelberg, 2009. Sp nger-Verlag.

37 J. Porubän and M. Bačíková. Definition of computer languages via user interfaces. In *FEEI 2010 : Electrical Engineering and Informatics : Proceeding of the FEEI of the TU of Kosice. Available online: http://hornad.fei.tuke.sk/~bacikova/publications/2010-08_feei10_kreutzova_CLANOK_final.pdf*, pages 53–57. Technical University of Košice, September 2010.

38 J. Porubän, Forgáč M., M. Sabo, and M. Běhalek. Annotation based parser generator. *Computer Science and Information Systems : Special Issue on Advances in Languages, Related Technologies and Applications*, 2010.

39 J. R. G. Pulido, S. B. F. Flores, R. C. M. Ramirez, and R. A. Diaz. Eliciting ontology components from semantic specific-domain maps: Towards the next generation web. In *Proceedings of the 2009 Latin American Web Congress (la-web 2009)*, LA-WEB '09, pages 224–229, Washington, DC, USA, 2009. IEEE Computer Society.

40 T. T. Quan, S. Ch. Hui, A. Ch. M. Fong, and T. H. Cao. Automatic generation of ontology for scholarly semantic web. In *International Semantic Web Conference'04*, pages 726–740, 2004.

41 Christopher Ramming, Mm. Calton, Pu Examinateurs, Renaud Marlet, and Charles Consel. Domain-specific languages: Conception, implementation and application, phd. thesis.

42 J. Seidenberg and A. Rector. Web ontology segmentation: analysis, classification and use. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 13–22, New York, NY, USA, 2006. ACM.

43 J. Shim and H. Lee. Automatic ontology generation using extended search keywords. In *Proceedings of the 2008 4th International Conference on Next Generation Web Services Practices*, NWESP '08, pages 97–100, Washington, DC, USA, 2008. IEEE Computer Society.

44 S. Sie and J. Yeh. Automatic ontology generation using schema information. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06, pages 526–531, Washington, DC, USA, 2006. IEEE Computer Society.

45 M. Simos and J. Anthony. Weaving the model web: a multi-modeling approach to concepts and features in domain engineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 94–102, 1998.

46 N. Souchon and J. Vanderdonckt. A review of xml-compliant user interface description languages. pages 377–391. Springer-Verlag, 2003.

47 R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures: Cdrl a011a curriculum module in the sei style. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, December 1995.

48 S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to omplementation application to video device drivers generation. conception, implementation and application. *IEEE Transactions on Software Engineering*, 25(3):363–377, 1999.

49 Q. T. Tho, S. Ch. Hui, A. C. M. Fong, and T. H. Cao. Automatic fuzzy ontology generation for semantic web. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):842–856, June 2006.

50 T. Thum, Ch. Kastner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200. IEEE, August 2011.

51 Y. A. Tijerino, D. W. Embley, D. W. Lonsdale, Y. Ding, and G. Nagy. Towards ontology generation from tables. *World Wide Web*, 8(3):261–285, September 2005.

52 K. Tilly and Z. Porkoláb. Semantic user interfaces. *IJEIS*, 6(1):29–43, 2010.

53 T. Tudorache, N. F. Noy, S. M. Falconer, and M. A. Musen. A knowledge base driven user interface for collaborative ontology development. In *Proceedings of the 16th international conference on Intelligent user interfaces*, IUI '11, pages 411–414, New York, NY, USA, 2011. ACM.

54 A. Valerio, G. Succi, and M. Fenaroli. Domain analysis and framework-based software development. *SIGAPP Appl. Comput. Rev.*, 5:4–15, September 1997.

55 O. Vasilecas, D. Kalibatiene, and G. Guizzardi. Towards a formal method for the transformation of ontology axioms to application domain rules. *Information Technology and Control*, 38(4):271–282, 2009.

56 M. Wimmer. A meta-framework for generating ontologies from legacy schemas. In *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application*, DEXA '09, pages 474–479, Washington, DC, USA, 2009. IEEE Computer Society.

57 T. Wong, W. Lam, and E. Chen. Automatic domain ontology generation from web sites. *J. Integr. Des. Process Sci.*, 9(3):29–38, July 2005.

58 H. Yang and J. Callan. Ontology generation for large email collections. In *Proceedings of the 2008 international conference on Digital government research*, dg.o '08, pages 254–261. Digital Government Society of North America, 2008.