Cent. Eur. J. Comp. Sci. • 4(4) • 2014 • 204-221
DOI: 10.2478/s13537-014-0210-9

VERSITA

Central European Journal of **Computer Science**

DSL-driven generation of Graphical User Interfaces

Research Article

Michaela Bačíková*, Jaroslav Porubän†

Technical university of Košice, Letná 9, 042 00, Slovakia

Received 28 February 2014; accepted 21 August 2014

Abstract: Domain-specific languages (DSLs) are becoming more and more popular. However, the number of DSLs is still small when comparing to the number of existing applications. The results of our previous research showed that it is possible to speed up the DSL development process by aiding the first development phases (design and implementation). More specifically it is possible to *generate* DSLs from existing GUIs (Graphical User Interfaces) of component-based applications. Now we want to use the generated DSLs and their models to generate new user interfaces or even whole new applications. To verify this claim, in this paper we use existing technologies which simplify the creation of web applications: iTasks. We also describe stereotypes of creating GUIs which we used to extract data from existing applications and to generate new applications. In the last part of this paper we limit the types of applications, which can be used for extraction based on our experiments with the prototype.

Keywords: DEAL • domain analysis • domain-specific languages • graphical user interfaces

1. Introduction

Domain-specific languages (DSLs) are computer languages tailored to a specific application domain [1–4]. DSLs provide language features tailored to a particular area of interest in the form of concrete syntax that incorporates notations from the domain and semantic analysis and optimizations that are typically not easily achieved in general purpose languages [5]. When comparing to general purpose languages (GPLs) (such as Java or C#), DSLs are more expressive and are easier to use while GPLs tend to be more general and are designed for solving problems in any domain. According to the empirical study of Kosar et al. [6] program understanding, in terms of learn, perceive and evolve, is much better for DSL programs than for GPL programs. DSLs are superior to GPLs in all cognitive dimensions that Kosar et al. define in their paper: closeness to mappings, diffuseness, error-proneness, role expressiveness, viscosity.
Currently, DSLs are becoming more and more popular. However, the number of DSLs is still small when comparing to the number of existing applications. Despite their advantages, the cost of developing a new DSL is usually high [1] because it involves development of language parsers and generators along with the language. The implementation phase of DSL development is well documented by many researchers such as Mernik [2] but the analysis and design phases are still dropped behind. The various DSL development phases and the tool support of each of them is described in the article by Čeh et al. [7]. According to Čeh et al., the phases of a DSL development are:

* *E-mail: michaela.bacikova@tuke.sk (Corresponding author)*
† *E-mail: jaroslav.poruban@tuke.sk*

1. Decision
2. Domain analysis
3. Design
4. Implementation
5. Testing
6. Deployment
7. Maintenance

In our work we try to support the first phases of DSL development, more specifically the domain analysis, design and implementation phases. In 2013 [8] we showed that it is possible to speed up the DSL development process by providing ways to generate a first prototype of a DSL, which can be then extended with specific features. This DSL prototype can help DSL developers to develop DSLs more quickly by providing a basic implementation of a DSL (a draft) to start with. Unlike the current approaches, we use a unique method of analysing *existing component-based user interfaces* as sources of domain information. We experimentally confirmed that it is possible to *generate DSLs from existing GUIs of component-based applications*.
In this paper we claim that based on these *generated DSLs* and their models, it is possible to generate *new user interfaces* or even whole new applications. To verify this claim we use existing technologies that simplify the creation of web applications: iTasks. We performed an experiment in which the DSLs and their models are automatically generated by our DEAL tool [9]. We implemented a new module into DEAL, which automatically generates new iTask applications based on a previously generated DSL. Through this experiment we aim to prove the possibility of generating a new UI based *solely on domain knowledge* that was extracted from an existing application. This concept of a "domain aware architecture" makes sense: after migrating to a new platform or a technology, the thing that with highest probability remains the same in the old and new applications will be their *domain*. By our approach, we try to preserve the domain model of an existing application for the purposes of further reuse in new application versions.
In addition, in this paper we describe stereotypes of creating graphical user interfaces which are used to extract data from existing applications and to generate new applications. Some stereotypes were identified in our previous works, some were identified during the work with the iTask system.
We realize that not every existing application can be used to extract domain information in the manner we describe in this paper. Therefore, in the last part of our paper we limit the types of applications, which can be used for extraction to ensure the highest possible degree of automation and the lowest amount of manual work needed after the generation process is finished.

1.1. Tasks and goals

In our previous research we strived for verifying the validity of the following hypothesis:

(H1) It is possible to extract a DSL from the interface of an existing component-based application in an automatized manner.

The validity of this hypothesis was shown on an experiment [8]. In this paper, we will define a new hypothesis based on the previous one:

(H2) It is possible to automatically generate a new GUI (or even a whole new application) from the previously extracted DSL.

A question is related both to hypothesis H1 and H2:

(Q1) To what extend it is possible to meet the hypotheses H1 and H2? How relevant are the results?

We try to satisfy the primary goal of this paper, which is *to show the validity of the hypothesis H2 and to answer the question Q1*, that means to define the boundaries, for which H1 and H2 are fulfilled. We assume the results will be strictly dependent on the application to be analysed (target application), namely on the following criteria:

- the programming style of the target application,
- the presence of components in the GUI of the target application,

- the types of components in the GUI of the target application,
- the presence of domain terms in the GUI of the target application.

Based on the defined goals the following tasks arise:

- To design and implement a *generator* of GUIs. The input of the generator will be a DSL generated by the DEAL method and the output will be a new generated GUI.
- To perform a series of *experiments* on existing component-based applications using the implemented generator.
- To evaluate the results and to create a list of *guidelines and restrictions for applications*, which can be used for the extraction and transformation process of DEAL.

2. The DEAL method

We have performed a research in the area of *automated domain analysis of UIs*, described in detail in [10] and [11]. Our method for 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 (introspection), reflection and/or aspect-oriented programming. The output of DEAL is a domain model and, consequently, a DSL of the target application. The DEAL traversal algorithm was described in [11] and in [8] we introduced the method of the GUI → DSL transformation.
The domain content of the target UI is extracted as a graph of *terms*, their *relations* and *properties*. We experimentally confirmed the possibility of extracting from Java and HTML applications [11]. Both languages meet the presumption to have the component nature and provide the possibility to determine the component structure. We also experimentally confirmed the possibility of extracting a DSL based on the extracted domain model [8] with existing Java applications.

3. Domain metamodel definition

The domain model holds the domain information extracted from an existing UI. Let us describe our domain metamodel using the following grammar[1]:

$$DomainModel \rightarrow Name\ Term\ SceneRef$$

As can be seen in the first grammar rule, each domain model (DM) has its *Name*, which can be derived from the scene name. The DM also has the reference to the *Scene* which it was created from and it contains one root *Term*.
Each *Term* can be of two types. An *Atomic* Term is a "leaf" term and a *Composite* term is a Term which can contain one ore more child Terms. Each *Atomic* Term can have a list of *Constraints*. Each *Composite* term can define a *Relation* between its child Terms. Both of the Term types contain information about their *Name*, *Description*, *Icon* and both have a reference to the component, which the Term was created from (target component):

$$\begin{aligned} Term &\rightarrow Composite\ \mid Atomic \\ Atomic &\rightarrow Name\ Description?\ Icon?\ Constraint^{*}\ ComponentRef \\ Composite &\rightarrow Name\ Description?\ Icon?\ Relation\ Term{+}\ ComponentRef \end{aligned}$$

[1] *In this work, when using grammars to explain language structures and models, non-terminals will be noted with first upper-case letter and terminals will be noted using quotes: "". Special terminals, such as data types and predefined values or terminals representing final abstract structures will be noted in ⟨⟩ brackets, e.g. ⟨STRING⟩ represents any terminal of type string (a string literal).*

The *Name* and optional *Description* represent the primary and secondary domain information extracted from the target component:

$$
\begin{aligned}
Name &\rightarrow \langle \text{STRING} \rangle \\
Description &\rightarrow \langle \text{STRING} \rangle
\end{aligned}
$$

Sometimes *icons* also contain domain information - whether it is text encoded in the icon, or information encoded in a graphical image. That is why the terms contain Icons as domain-specific information:

$$Icon \rightarrow \langle \text{ICON} \rangle$$

Each *Composite* term contains information about the *Relation* between its children:

$$
\begin{aligned}
Relation \rightarrow \; & \langle \text{MODEL} \rangle \mid \langle \text{AND} \rangle \mid \langle \text{MUTUALLY\_EXCLUSIVE} \rangle \mid \\
& \langle \text{MUTUALLY\_NOT\_EXCLUSIVE} \rangle
\end{aligned}
$$

There are 4 types of *relations* in the DM:

- MODEL – it represents "and" relation and it also indicates that the term is a root node
- AND – the children of this term are in an "and" relation.
- MUTUALLY_EXCLUSIVE – the children of this term are mutually exclusive.
- MUTUALLY_NOT_EXCLUSIVE – the children of this term are not mutually exclusive.

Each *Atomic* term can have a set of constraints limiting its use:

$$
\begin{aligned}
Constraint &\rightarrow DataTypeConstraint \mid Enumeration \mid Length \mid \\
& \quad\; Range \mid Regex \\
DataTypeConstraint &\rightarrow \langle \text{STRING} \rangle \mid \langle \text{REAL} \rangle \mid \langle \text{NUMERIC} \rangle \mid \langle \text{DATE} \rangle \\
Enumeration &\rightarrow \langle \text{STRING1} \rangle \langle \text{STRING2} \rangle \ldots \langle \text{STRINGn} \rangle \\
Length &\rightarrow Min\ Max \\
Range &\rightarrow Min\ Max \\
Regex &\rightarrow \langle \text{STRING} \rangle \\
Min &\rightarrow \langle \text{INTEGER} \rangle \\
Max &\rightarrow \langle \text{INTEGER} \rangle
\end{aligned}
$$

The constraints can be:

- *DataTypeConstraint* – represents the term's data type: String, Real, Numeric or Date.
- *Enumeration* – represents a set of term's possible values.
- *Length* – limits the maximum and minimum length of the term's value (only for String types).
- *Range* – limits the maximum and minimum range of the term's value (only for Numeric types).
- *Regex* – the values of the term have to comply to the pre-defined regular expression.

The domain metamodel defines the outputs of the DEAL method (domain models).

4. The DEAL method phases

The DEAL method can be divided into 5 phases displayed in Fig. 1 along with the output of each phase. The input of DEAL is the target application UI. The output is a domain model, which is further used in other processes.

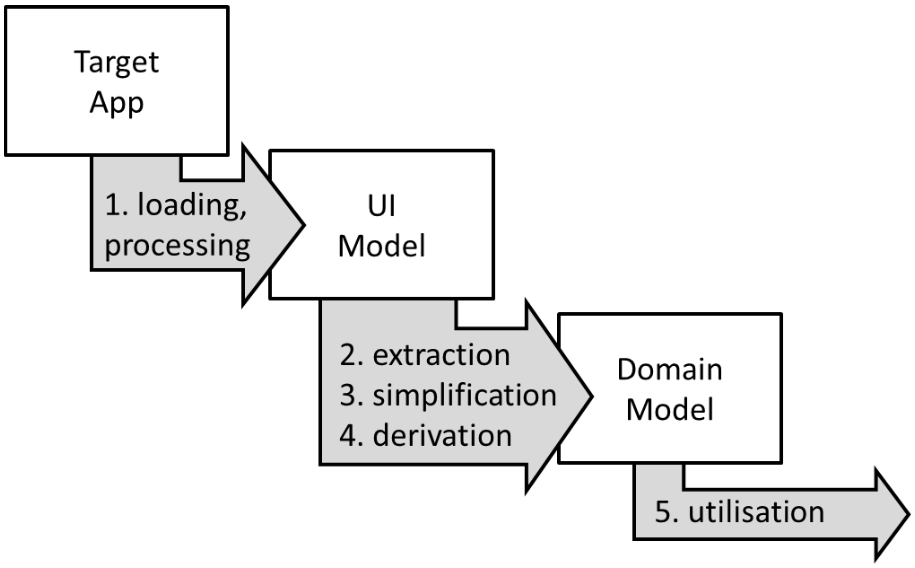


Figure 1. The phases of the DEAL method.

1. Loading and processing

The target application is loaded into memory and all its scenes and components are traversed and processed. The loading and processing phase is application-specific and the programming language of the target application has to provide means of introspection for this phase to be successful. Application-specific means that a new loading and processing algorithm has to be created for every new programming language of the target application. The output of the loading and processing phase is a an application UI model.

2. Extraction

The domain model is extracted from the application's UI using the UI model created in the previous phase. The extraction phase is component-specific. Component-specific means that a new handler has to be created for each new component type and programming language. The output of the extraction phase is an object representation of the domain model, which contains terms and their explicit properties, constraints and relations. The algorithm was described in our previous works.

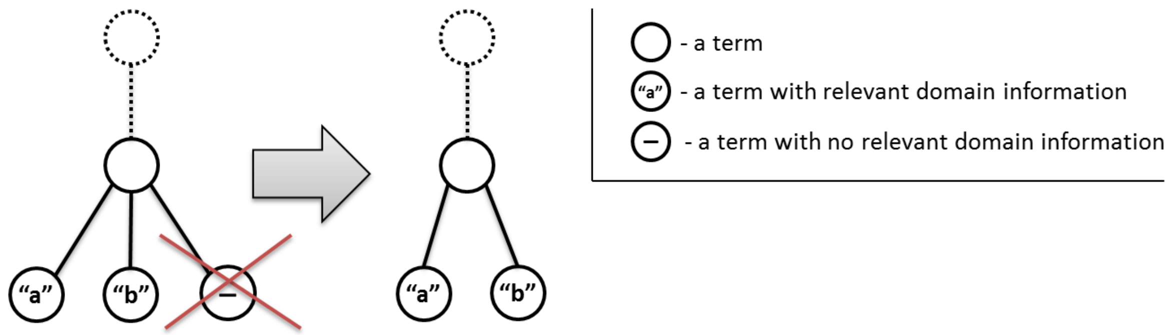

"a"
"b"
–
"a"
"b"
- a term
"a" - a term with relevant domain information
– - a term with no relevant domain information

Figure 2. Removing of empty leafs - if there is any leaf, which does not contain any relevant domain information, it will be removed.

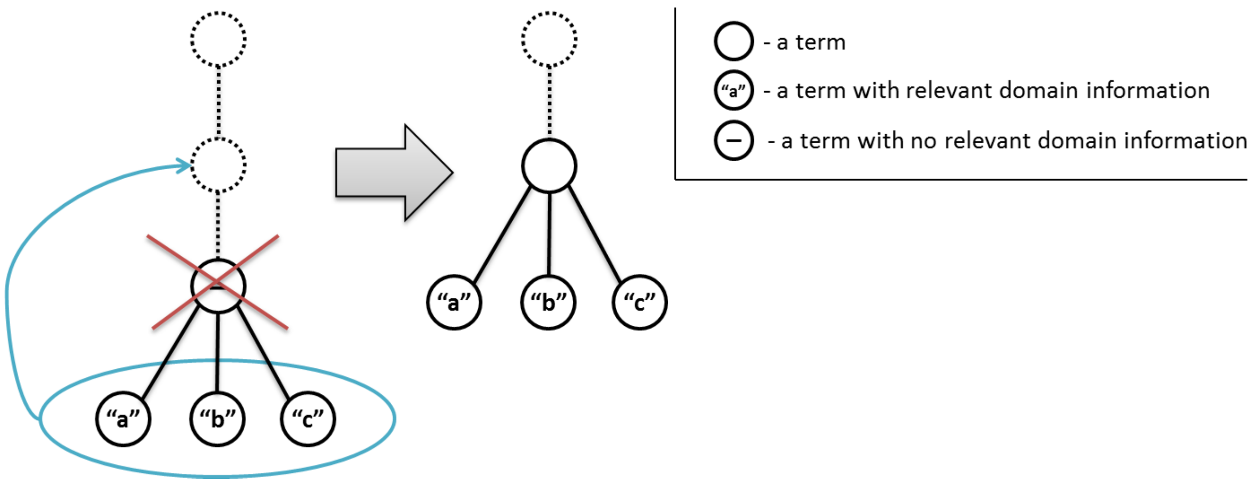

–
"a"
"b"
"c"
"a"
"b"
"c"
- a term
"a" - a term with relevant domain information
– - a term with no relevant domain information

Figure 3. Removing void containers and multiple nesting - if there is a composite term with no relevant domain information and its parent contains only this one term, then the composite term will be removed and its children will be moved to the parent.

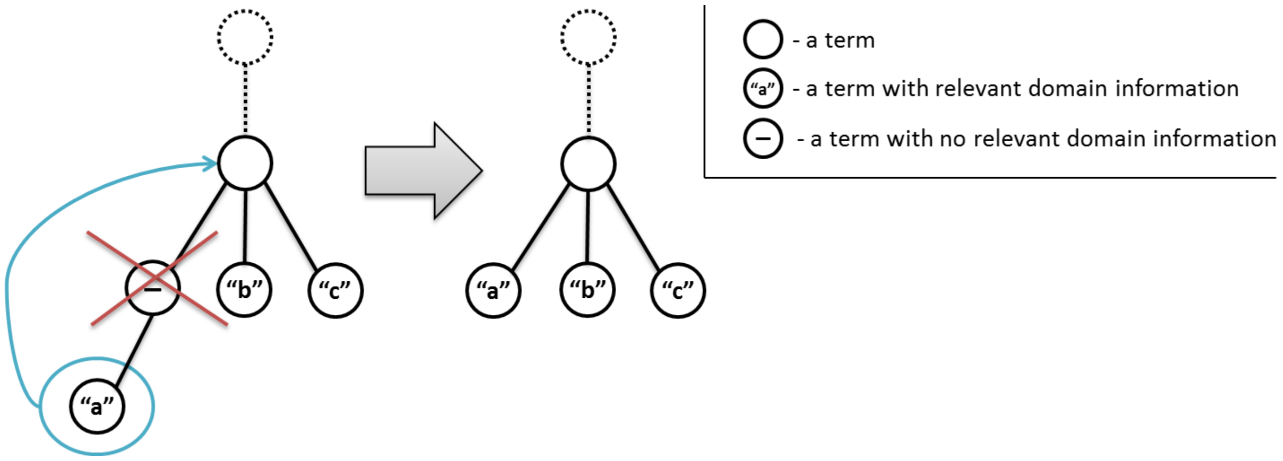

–
"b"
"c"
"a"
"a"
"b"
"c"
- a term
"a" - a term with relevant domain information
– - a term with no relevant domain information

Figure 4. Shifting single leafs - if there is any composite containing only a single leaf child, the composite will be removed and the child will be moved to the parent of the removed composite.

3. Simplification

Filtering information unrelated to the domain. The domain model created in the extraction phase contains data not important for the domain model, such as empty terms (terms without any relevant domain information, created mostly from containers), general application-specific terms, etc. Filtering involves removing multiple nesting (Fig. 2), removing void containers and multiple nesting (Fig. 3), shifting single leafs (Fig. 4) and filtering out general application-specific terms. A void container is a term created from a component of type Container that does not contain any children. Some empty terms are essential for preserving the hierarchy of terms. However if the target application contains too many containers, it causes excessive deepening of the domain model hierarchy, which is not desirable. Therefore such terms have to be removed.
The output of the simplification phase is a simplified domain model.

4. Derivation

Derivation of implicit relations. In DEAL, relations defined on parent terms define the relation between their child terms. Some relations can be directly derived from the component type (such as a list component with a single selection, where the items of the list are mutually exclusive). However, for some component types (such as tab panes in Java language or more common check box buttons), the relation can be determined only when evaluating the children of the parent. Such relations are derived in this phase. Derivation is defined in extraction handlers and is based on the identification of different types of components. The output is a simplified domain model with relations between terms.

5. Utilisation

Further usage of the model created in the previous phases in other processes such as generating a DSL [8], generating an ontology [12], generating new user interface, evaluating domain usability [13], etc.

These phases are to be performed sequentially in the order they are listed here. Each phase uses the output of the previous phase. Once the domain model is extracted, it can be processed independently. Therefore the simplification, derivation and utilisation phases are not necessarily application-specific: they are the same for any target application.

5. The DEAL tool prototype

The DEAL tool prototype[2] is an implementation of the DEAL method on Java applications. Currently, DEAL uses YAJCo[3] language processor to generate grammars. More about YAJCo can be found in the publication by the authors of the tool Porubän et al. [14].
The DEAL tool prototype proves that it is possible to:

- traverse the GUI of an existing Java application,
- extract domain information in a formalized form (domain model),
- generate a DSL grammar, model, and parser based on the domain information.

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

[2] *DEAL project can be found at: https://www.assembla.com/spaces/DEALtool along with tutorials and references.*
[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*

An example of the DEAL output for the person form (Fig. 5) is the grammar generated from the extracted model:

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

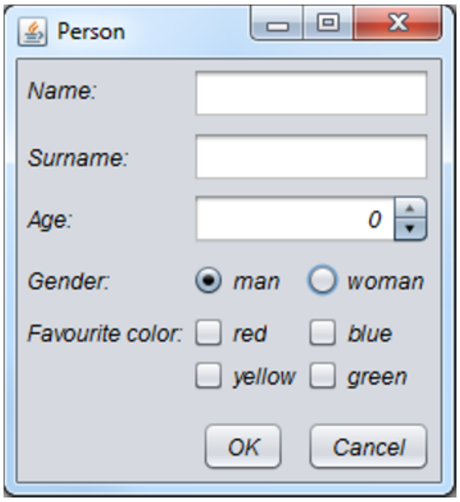


Figure 5. An example of a person form.

The grammar is in the YAJCo notation and the rules for not-mutually exclusive terms (colors) are supplemented by the $0 - n$ version because YAJCo tool does not support the "?" operator. However the results show that generating DSL grammars from existing UIs is definitely possible. Along with the grammar, YAJCo tool generates a JavaCC parser for the DSL.

6. DSL → GUI Transformation

The transformation process is described in Fig. 6. The DEAL tool runs an existing Java application, traverses its components and extracts its domain model and a DSL. The DSL is available in an object model and it is created by directly transforming domain concepts in the DEAL domain model into language concepts. Based on the extracted DSL, a code for a new application is generated. The generated code is written in iTask and the GUI of the resulting application is created automatically by the iTask system. To generate iTask code we used direct transformation, the rules for which is described in the section 6.3. The functionality for generating iTask code was integrated into the DEAL tool for the purposes of this paper.

6.1. Why iTasks?

The iTask system (iTasks) is a task-oriented programming toolkit for programming workflow support applications in Clean [15]. Workflows consist of *typed tasks* that produce results that can be passed as parameters. From these iTask specifications, executable workflow systems - new functional GUIs - are automatically generated by the iTask system. We will explain the principle of iTasks on an example of an iTask code for a Person form (see listing 1).

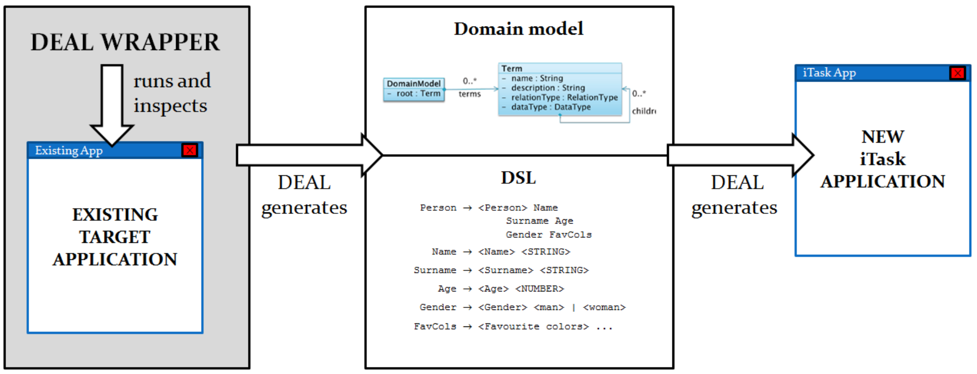


Figure 6. The process of the GUI → DSL → GUI transformation.

Listing 1. Example of a Person form written in iTask

```
:: MyPerson =
          { name                          :: String
          , surname                       :: String
          , age                           :: Int
          , gender                        :: MyGender
          , favouriteColors               :: MyColors
          }

:: MyGender = Male | Female
:: MyColors = { blue::Bool, white::Bool,
                yellow::Bool, green::Bool}
derive class iTask MyPerson, MyGender, MyColors

enterPerson :: Task MyPerson
enterPerson = enterInformation
                    "Enter your personal information:" []
```

In listing 1, the model of the application is represented by the `MyPerson` type along with its subtypes `MyGender` and `MyColors`. The **`derive class`** command creates the classes of `MyPerson`, `MyGender` and `MyColors` so they can be used in the subsequent code. The last two rows define a new task `enterPerson`. After adding the `enterPerson` task to the iTask System workflow, the iTask system automatically generates a web form displayed in Fig. 7. The iTask system also creates means of checking user input (the green icons in Fig. 7).
As can be seen in the example, the same relations and properties extractable by DEAL can also be expressed in iTask:

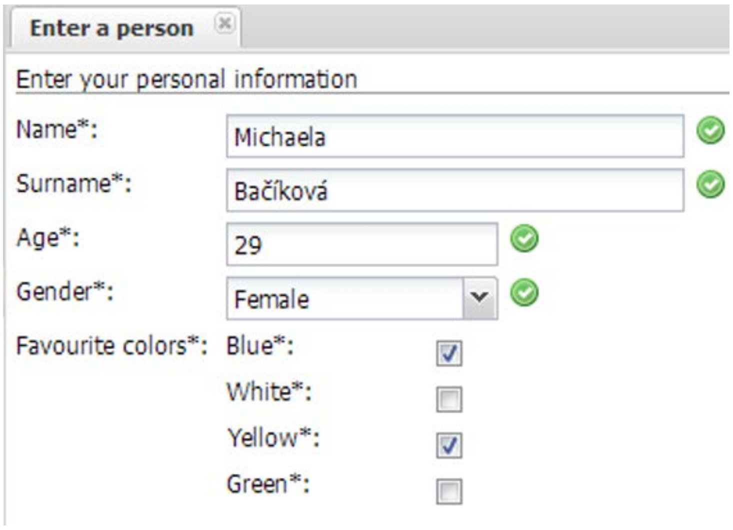
- *data types*: expressed by the task type or by the data type property,
- *mutually-exclusive relation*: expressed by alternatives (e.g. in the MyGender type),
- *mutually-not-exclusive relation*: expressed by a Set (e.g. in the MyColors type).

Similarly to the `enterPerson` task, a task for viewing a person can be implemented using the code described in listing 2.

Listing 2. Person view example in iTask

```
viewPerson :: Task MyPerson
viewPerson = viewInformation "This is the person:" []
```

The provided examples clearly show the simplicity of writing a simple entry (or display) form only by providing a domain model and a few lines of additional code. Now we are only a small step from generating iTask applications: the domain model is automatically provided by DEAL, its notation only has to be transferred into the iTask notation.


Enter a person

Enter your personal information

Name*:	Michaela	✓
Surname*:	Bačíková	✓
Age*:	29	✓
Gender*:	Female	✓
Favourite colors*:	Blue*:	☑
	White*:	☐
	Yellow*:	☑
	Green*:	☐

Figure 7. The example of a person form generated by the iTask System.

6.2. Identified stereotypes

In our previous paper [11] we identified a number of stereotypes of creating UIs. Heuristic transformation rules for generating DSLs were later created based on these stereotypes [8]. In this section we will summarize all previously identified stereotypes and introduce a few new, identified during the work with the iTask System. Based on this list, we will define new transformation rules in the future, and then we will include them in the DEAL prototype. The list of stereotypes follows:

1. *Hierarchical arrangement* of components is usually very similar to (or the same as) the hierarchy of entities and properties in the domain model.
2. *Labels* have two functions:
   - determine the purpose of other components by labelling them,
   - display data.
3. For data entry, usually *textual*, or other components are used. They can be of different types and they can have restrictions depending on the type of information entered (or it's length).
   (a) Standard *text fields* are used to enter information of type STRING.
   (b) To enter information of type STRING with greater length and containing new lines, *text areas* are usually used.
   (c) To enter numeric values, *spinners* or *text fields with number restriction* are used.
   (d) To enter passwords, *encoded text fields* are used.
   (e) To enter a single selection from multiple options, *combo boxes*, *lists* and *radio buttons* are used.
   (f) To enter a multiple selection from multiple options, *lists* and *check boxes* are used.
   (g) To enter dates, *spinners*, *text fields with restriction to dates* and *special components of type calendar* are used.
   (h) To enter a geographical location a *map component* is used (e.g. Google Maps).
   (i) To enter a selection from a tree structure, a *tree component* is used.
   (j) To restrict the data, restrictions on components are set:
      - upper and lower limits,
      - a list of values.
4. Forms usually contain *OK*, *Cancel* and *Reset buttons*, which represent the functionality of form confirmation, cancelling of entering values or cleaning up the form values.

5. Applications, which provide an additional functionality, use additional functional components such as *buttons* or *menu items*.
6. *Buttons* are usually located in dialogs or windows and the result of clicking on a button is an execution of a functionality, which is predefined on the button in the code and a possible change of the application state.
7. *Menu items* are usually structured in menus and similarly to buttons, the result of clicking on a menu item is an execution of the predefined functionality and a possible application state change.
8. Functional components can trigger opening a new window/dialog or closing an existing window/dialog.
9. *Tab panes* are used to structure window content. Each tab has its name which identifies a sub-domain and all items located in the particular tab belong to this sub-domain.

6.3. Transformation to the iTask Code

To express some of the stereotypes mentioned above, iTasks contain the following predefined structures:

String	– for textual information,
Int	– for numeric information,
Date	– calendar for entering dates,
Note	– for a long textual information,
"|" operator	– for a single selection,
Set	– to represent a list of values for multiple selection or to restrict the minimum or maximum value of data,
GoogleMaps	– for geographical location,
Programmer defined types	– to express structured information.

The iTask code is based on defining DSLs. DSLs are formed by specifying classes and their properties. Based on the type of any DSL property (e.g. String, Int, Date, Note, Set, etc.), a corresponding component is generated. The transformation of the types to graphical components in iTasks is as follows:

String	$\longrightarrow$	a text field,
Int	$\longrightarrow$	a spinner (a numeric text field),
Date	$\longrightarrow$	a calendar for entering dates,
Note	$\longrightarrow$	a text area,
"|" operator	$\longrightarrow$	a list (a combobox) with a single selection,
Set	$\longrightarrow$	a group of check-box buttons or a text field with checked maximum and minimum values,
GoogleMaps	$\longrightarrow$	a Google map component,
Programmer defined types	$\longrightarrow$	programmatically defined component

In DEAL, for each grammar rule of DSL grammar, one *class definition* is created in iTask (e.g. `MyPerson` in listing 1) and for each non-terminal in the DSL grammar, one *property definition* in iTask is created (e.g. `name :: String` in listing 1). In case of alternatives (a mutually not exclusive relation), a *set* is generated in iTask (e.g. the `MyColors` class in listing 1). In case of an or rule (a mutually exclusive relation), an *or class* is generated in iTask (e.g. the `MyGender` class in listing 1). The types of the DSL properties are directly transformed into the iTask code. For each new DSL, two tasks are created and added to the iTasks workflow: an `enterInformation` and a `viewInformation` task.
The layout of the resulting forms is automatically created by iTasks, along with default buttons (if they are necessary). In addition, iTasks also provides a possibility to add programmer defined functionalities by creating *Action* tasks, results of which are menu items or buttons. *OK*, *Cancel* or *Reset* buttons are generated automatically, if the form requires them. The semantics of the graphical components is defined by their functionality. This is the reason why we can not extract the semantics of each component by our method. The functionality of commonly used actions, such as *OK*, *Cancel*, *Reset*, *Save* or the semantics of standard input components such as text field, check box or radio button, can be predicted and automatically generated into the iTask code. However, functionalities defined by the UI developer cannot be directly and automatically transformed into the iTask code without a deep source code analysis. The GUIs created by our module are not final, they represent partially functional prototypes and the transformation is aimed at DSL reuse. Therefore we

do not define transformation for developer-specific components. To reproduce such specific functionalities, the developer has to be familiar with the Clean language and with the iTask system and manually enter the functionality of her/his specific components into the generated application. In addition to the DSL transformation, we also transform menu item components into the iTasks. For each menu item in the source application, a new menu item in the resulting application is created. However for the here mentioned reasons, the functionality of the menu items is not implemented.

7. Experiments and results

We created a prototype of the transformation module described in section 6 for the purpose of experimental verification of the hypothesis H2 defined in the introduction. The transformation module was integrated into the DEAL prototype. The input of the transformation module is a DSL and domain model generated by DEAL. The functionality of extracting DSLs from the existing application GUIs was already implemented in the DEAL prototype, the module for transforming the DSLs into the iTask code is new. The output of the module is an iTask application. We verified the transformation module on all 17 Java applications which are included in the DEAL prototype. The results can be summarized as follows:

- For the standard components, the transformation produced expected results.
- If the target application contained non-standard components, they were not extracted and therefore they were not present in the generated iTask application. Additional programming would be needed to include the support for non-standard components into DEAL.
- Although we were able to extract and generate all buttons and menus, the functionality of user defined buttons and menus could not be extracted and transformed into the iTask application. Such functionalities have to be manually programmed into the generated application code.
- Components without any identifier (i.e. label or tool-tip) were not extracted. If they represented a grouping node, a term with a randomly generated name (e.g. "Unknown123456") was created instead.
- The best results were produced from applications, which were simple forms, with a properly defined terminology and no special functionalities and components.

Based on the listed results we can conclude, that the highest possible degree of automation and the lowest amount of manual work after the generation process is finished can be achieved by selecting the target application which would:

- contain only standard components,
- have properly defined domain model,
- look like a simple form or dialog (or set of forms and dialogs) with standard functionalities (OK, Cancel, Reset).

The above defined characteristics represent recommendations for applications, which can be used as an input of DEAL in order to be used for generating new, functional UIs. Other applications can be analysed and transformed too, but much more manual work will be needed after the transformation. For each specific functionality, which is not defined in iTasks, new specific code is needed in the generated application code. To be able to create new functions, knowledge of the Classic language and iTask system is required.

8. Related work

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

8.1. Domain analysis

The domain analysis (DA) was first defined by Neighbors [16] in 1980 and he stresses that DA 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. *DREAM* (Domain Requirements Asset Manager) approach by Mikeyong et. al. [18] is similar to FODA, 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 [19], CaptainFeature[4], RequiLine[5] or ASADAL[6]. Other examples of formal methodologies are ODM (Organization Domain Modeling) [20] and DSSA (Domain Specific Software Architectures) [21].

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* [22] that uses a set of libraries; or the later *Sherlock* environment by Valerio et al. [23] 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 concrete form of a model depends on its purpose and level of abstraction. The more abstract the model is, the higher is the shielding of the overall code complexity. Models are used to increase productivity. Modelling is often closely connected with DSLs. The MDD principle support is provided for example by the Czarnecki project Clafer [24] and the FeatureIDE plug-in [25] by Thüm and Kästner.

ToolDay (A Tool for Domain Analysis) [26] 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 [27] 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 necessarily have to contain the domain terms and domain processes. The DARE tool *does not analyse the GUIs* specifically.

Ontologies as sources of domain analysis

Last but not least, the approach most similar to ours is the one proposed by Čeh et al. [7]. 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 in ontological databases when comparing to the amount of existing software systems.

The latest work in the area of ontology to DSL transformation is the one by Tairas et al. [28]. In this work the authors show a practical usage of an ontology to create a new DSL in the domain of Air Traffic Communication. Although the problem of non-existent ontology is still evident in this paper (the authors had to manually create their own ontology for the purposes of creating a DSL), the paper also touches feature modelling, i.e. analysing commonalities and variabilities of domain concepts. The knowledge of commonalities and variabilities of the domain concepts can further provide crucial information needed to determine the fixed and variable parts of the language [28] and this could be a significant extension of our approach. However, this problem is still open also in the work of Tairas et al.

The ontological format also has its advantages. In our approach, a new domain model is created for every scene in a UI and the concepts in the resulting models are not interconnected. For example, let us have a dialog in the JabRef[7] application, which provides a selection of different types of bibliographical references: article, proceedings, book, etc. And the main JabRef window contains secondary information about a single reference, e.g. a title, authors, date, etc. After the extraction with DEAL, we have two domain models which are not interconnected. E.g. it is not clear that a reference of type article can have a booktitle. To create this reference, we need language composition. To our knowledge, there is no automated support for language composition in the area of DSLs.

[4] *CaptainFeature, the webpage of CaptainFeature SourceForge.net project, `http://captainfeature.sourceforge.net/`, 2005*

[5] *The webpage of RequiLine project, `https://www2.swc.rwth-aachen.de/RequiLine`, 2005*

[6] *A review of ASADAL CASE tool, Postech Software Engineering Laboratory, `http://selab.postech.ac.kr/asadal`, 2011*

[7] *http://jabref.sourceforge.net/*

Since ontologies are a standard, there is a wide support for working with ontologies and one of them is *ontology merging* that could be used for composing two models. For this purposes we developed an approach of generating ontologies from our domain models in [12]. This way, the ontologies created by our DEAL method could be used in the *ontology to DSL* transformation described by Čeh et al. and Tairas et al.

8.2. Ontology extraction

Many approaches are targeted to ontology learning. 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:

i) clustering of terms [29, 30],
ii) pattern matching [29–32],
iii) heuristic rules [30–32],
iv) machine learning [33],
v) neural networks, web agents, visualizations [30],
vi) transformations from obsolete schemes [32],
vii) merging or segmentation of existing ontologies [29, 34],
viii) using fuzzy logic to generate a fuzzy ontology, which can deal with vague terms such as FFCA method [35] or FOGA method [36],
ix) analysis of web table structures [31, 37],
x) analysis of fragments of websites [38].

8.3. GUI nodelling 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 [39], or modern languages, such as XML, described in the review made by Suchon and Vanderdonckt in [40]. Paulheim [41] designed UI models of interaction with users. For UI configurations usually models such as configuration ontology designed for WebProtégé tool in [42] are used.
The most complex UI model was designed by Kösters in [43] 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 [44].

8.4. 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 [45],
- generating of UIs based on models and ontologies by Kelshchev and Gribova in [46],
- deriving UIs from ontologies and declarative model specifications by Liu et al. in [47],
- program analysing and language inference [48].

A very interesting process is also seen in [49] where authors transform ontology axioms into application domain rules however the results are not as formal as our DSL grammar.

8.5. DSL and Grammar Inference

Grammatical inference can be used for grammar extraction from input examples as shown in [50] by Hrnčič et al. Their MAGIc (Memetic Algorithm for Grammar Inference) algorithm derives grammars using the combination of a population-based evolutionary approach and local search techniques. The initial population of grammars is not generated randomly, but it is generated from input examples. Each initial grammar parses one input example. Sabo et al. on the other hand perform a specification of computer languages using inference from program examples [51]. These approaches directly touch our approach. The examples of programs can be provided by domain experts and the grammars, or DSLs respectively, can be directly derived from the concepts and operators used in those examples. To be able to create a DSL of a high quality, a high number of examples is required.

In our approach, a similar technique could be applied. When a user inputs data into a UI form, (s)he practically creates *sentences in the GUI programming language*. If the GUI form is created with means of checking the input data (e.g. the format of dates is checked by the program), then the form can be used to evaluate the correctness of the sentences written by users (if the sentence is not correct, the OK button will not lead to a new scene and require the input again). The same it is with grammars and textual examples.

This correspondence between grammars, their parsers and GUIs is obvious even more when creating ontologies from existing GUIs [12]. The data provided by users represent ontology individuals and the form definition represents an ontology class. Individuals should correspond to classes.

Using the DEAL method, a set of examples can be automatically created from the inputs provided when using the GUI forms by users. Imagine a situation, where a personnel manager works with an application for managing company employees. Since it is a big international company, the personnel manager uses the application on a daily basis (i) to input data about new company employees; (ii) to change information about the existing employees; and (iii) to remove the fired employees from the company database.

Using our DEAL method implementation, running in background, we could track all the data submitted by the personnel manager. Not only that, we would also be able to check if the data were correct or incorrect. Let us assume, that when successfully entering a new employee (or when modifying an existing employee), the input form will be closed and the main screen of the application is displayed. When incorrectly entering an employee, the user is simply prompted to enter the information again. Based on the behaviour of the application after submitting an employee, we can detect if the data were correct (a new scene was opened) or incorrect (the same scene remains opened). This way we could automatically collect a set of examples of sentences along with the information about their correctness. These examples can then be used for grammar inference.

The advantage of such approach would be the natural way of entering information into a user interface, which the users use on a daily basis, while they are not disturbed from their ordinary work. To ask them for a textual notation of a DSL sentences is not only unnatural for users, it is also time-consuming - they entered the information into the system once, why should they write it again?

After creating a DSL from the GUI, the correctness of the sentences evaluated by the GUI form can also be evaluated by the DSL parser (which is automatically generated by DEAL, too). This way we would be sure of the DSL correctness. We think, that to infer a DSL grammar, not only correct, but also incorrect examples should be considered. The approach of comparing the results with an existing GUI would have means to evaluate both syntactically correct and syntactically incorrect examples and to modify the DSL grammar accordingly.

9. Conclusion

In our previous research we proved the possibility of generating DSLs from existing GUIs of component-based applications. Based on the DSLs, it is possible to generate new user interfaces or even whole new applications, which we showed in this paper. We designed, implemented and experimentally verified a DSL $\rightarrow$ GUI transformation module. As an input we used the outputs of our current project, DEAL: DSLs generated from UIs. To create the output, we used existing technologies which simplify the creation of web applications: iTasks. Through experimental verification we aimed to prove the possibility of generating a new UI based *solely on domain knowledge* extracted from an existing application. We also listed a number of previously identified and new stereotypes, which were used to create transformation rules for the DEAL extractor and generator. Although the generation process provided expected results for form-based applications, for more structured applications with special components and programmer defined functionality an additional manual work is needed after the generation process. In the last part of the paper we limited the types of applications, which can be used for extraction to ensure the highest possible degree of automation and the lowest amount of manual work after the generation process is finished. In the future we plan to improve DEAL to support more component types based on the stereotypes defined in this paper. Primary, the DEAL tool will be used to evaluate domain usability of existing user interfaces.

Acknowledgement

This work was supported by VEGA Grant No. 1/0341/13 Principles and Methods of Automated Abstraction of Computer Languages and Software Development Based on the Semantic Enrichment Caused by Communication, duration: 2013-2015, coordinator: prof. Ing. Ján Kollár, CSc.

References

[1] M. Fowler, Domain-Specific Languages (Addison-Wesley Signature Series (Fowler) (Addison-Wesley Professional, 2010)
[2] T. Kosar, P.E.M. López, P.A. Barrientos, M. Mernik, A preliminary study on various implementation approaches of domain-specific language, Inf. Softw. Technol. 50(5), 390–405, 2008
[3] J. Kollár, S. Chodarev, Extensible approach to DSL development, J. Inform. Control Management Syst. 8(3), 207–215, 2010
[4] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37(4), 316–344, 2005
[5] A. van Deursen, P. Klint, J. Visser, Domain-specific Languages: An Annotated Bibliography, SIGPLAN Not. 35(6), 26–36, 2000
[6] T. Kosar, N. Oliveira, M. Mernik, M.J.V. Pereira, M. Črepinšek, D. da Cruz, P.R. Henriques, Comparing General-Purpose and Domain-Specific Languages: An Empirical Study, Comput. Sci. Infor. Syst. 7(2), 247–264, 2010
[7] I. Čeh, M. Črepinšek, T. Kosar, M. Mernik, Ontology driven development of domain-specific languages, Comput. Sci. Infor. Syst. 8(2), 317–342, 2011
[8] M. Bačíková, J. Porubän, D. Lakatoš, Defining Domain Language of Graphical User Interfaces, SLATE (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik - place:Dagstuhl, Germany, 2013) 187–202
[9] M. Bačíková, Domain analysis with reverse-engineering for GUI feature models, POSTER 2012 : 16th International Student Conference on Electrical Engineering, 16, 1–5, 2012
[10] M. Kreutzová (Bačíková), J. Porubän, P. Václavík, First Step for GUI Domain Analysis: Formalization, J. Comput. Control Syst. 4(1), 65–70, 2011
[11] M. Bačíková, J. Porubän, Analyzing stereotypes of creating graphical user interfaces, Cent. Eur. J. Comp. Sci. 2(3), 300–315, 2012
[12] M. Bačíková, Š. Nitkulinec, Formalization of Graphical User Interfaces using Ontologies, POSTER 2014 : 18th International Student Conference on Electrical Engineering 15, 1–5, 2014
[13] M. Bačíková, J. Poruban, Human System Interaction (HSI), 2013 The 6th International Conference on Ergonomic vs. domain usability of user interfaces, Gdańsk, Poland, June 2013, 159–166
[14] J. Porubän, M. Forgáč, M. Sabo, M. Běhalek, Annotation based parser generator, Comput. Sci. Inform. Syst.: Special Issue on Advances in Languages, Related Technologies and Applications 7(2), 291–307, 2010
[15] J.M. Jansen, R. Plasmeijer, P. Koopman, P. Achten, Embedding a web-based workflow management system in a functional language, Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA '10 (ACM, New York, USA, 2010) 1–8
[16] J.M. Neighbors, Software construction using components (University of California, Irvine, 1980)
[17] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study (Carnegie-Mellon University Software Engineering Institute, 1990)
[18] M. Moon, K. Yeom, H. Seok Chae, An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line, IEEE T. Softw. Eng. 31(7), 551–569, 2005
[19] K. Czarnecki, T. Bednasch, P. Unger, U.W. Eisenecker, Generative Programming for Embedded Software: An Industrial Experience Report, Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming

and Component Engineering, GPCE '02 (Springer-Verlag, London, UK, 2002) 156–172

[20] M. Simos, J. Anthony, Software Reuse, 1998. Proceedings. Fifth International Conference on Weaving the model web: a multi-modeling approach to concepts and features in domain engineering (IEEE, 1998)

[21] R.N. Taylor, W. Tracz, 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, 1995

[22] R.P. Díaz, Reuse Library Process Model. Final Report (Electronic Systems Division, Air Force Command, USAF, Hanscomb AFB, MA, 1991)

[23] A. Valerio, G. Succi, M. Fenaroli, Domain analysis and framework-based software development, SIGAPP Appl. Comput. Rev. 5(2), 4–15, 1997

[24] K. Bak, K. Czarnecki, A. Wasowski, Feature and meta-models in Clafer: mixed, specialized, and coupled, Proceedings of the Third international conference on Software language engineering, SLE'10 (Springer-Verlag, Berlin, Heidelberg, 2011) 102–122

[25] T. Thum, Ch. Kastner, S. Erdweg, N. Siegmund, Software Product Line Conference (SPLC), 2011 15th International, Abstract Features in Feature Modeling (IEEE, Munich, Germany, 2011) 191–200

[26] L. Lisboa, V. Garcia, E. de Almeida, S. Meira, ToolDAy: a tool for domain analysis, Int. J. Software Tools Technol. Transfer (STTT) 13(4), 337–353, 2011

[27] W. Frakes, R. Prieto-Diaz, Ch. Fox, DARE: Domain analysis and reuse environment, Ann. Softw. Eng. 5, 125–141, 1998

[28] R. Tairas, M. Mernik, J. Gray, Using ontologies in the domain analysis of domain-specific languages, Models in software engineering: Workshops and Symposia at MODELS 2008, Reports and Revised Selected Papers (Springer, Berlin, Heidelberg, 2009) 332–342

[29] H. Yang, J. Callan, Ontology generation for large email collections, Proceedings of the 2008 international conference on Digital government research, dg.o '08 (Digital Government Society of North America, Montreal, Canada, 2008) 254–261

[30] J.R.G. Pulido, S.B.F. Flores, R.C.M. Ramirez, R.A. Diaz, Eliciting Ontology Components from Semantic Specific-Domain Maps: Towards the Next Generation Web, Proceedings of the 2009 Latin American Web Congress (la-web 2009), LA-WEB '09 (IEEE Computer Society, Washington DC, USA, 2009) 224–229

[31] Y.A. Tijerino, D.W. Embley, D.W. Lonsdale, Y. Ding, G. Nagy, Towards Ontology Generation from Tables, World Wide Web 8(3), 261–285, 2005

[32] M. Wimmer, A Meta-Framework for Generating Ontologies from Legacy Schemas, Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application, DEXA '09 (IEEE Computer Society, Washington DC, USA, 2009) 474–479

[33] B. Omelayenko, Learning of Ontologies for the Web: the Analysis of Existent Approaches, In Proceedings of the International Workshop on Web Dynamics (2011)

[34] J. Seidenberg, A. Rector, Web ontology segmentation: analysis, classification and use, Proceedings of the 15th international conference on World Wide Web, WWW '06 (ACM, New Yor, USA, 2006) 13–22

[35] W. Chen, Q. Yang, L. Zhu, B. Wen, Research on Automatic Fuzzy Ontology Generation from Fuzzy Context, Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation - Volume 02, ICICTA '09 (IEEE Computer Society, Washington, DC, USA, 2009) 764–767

[36] Q.T. Tho, S.Ch. Hui, A.C.M. Fong, T.H. Cao, Automatic Fuzzy Ontology Generation for Semantic Web, IEEE T. Knowl. Data Eng. 18(6), 842–856, 2006

[37] A. Pivk, Automatic ontology generation from web tabular structures, AI Communications 19, 2006, 2005

[38] T. Wong, W. Lam, E. Chen, Automatic Domain Ontology Generation From Web Sites, J. Integr. Des. Process Sci. 9(3), 29–38, 2005

[39] M.M. Moore, S. Rugaber, Domain Analysis for Transformational Reuse, Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97) (IEEE Computer Society, Washington DC, USA, 1997)

[40] N. Souchon, J. Vanderdonckt, A review of XML-compliant user interface description languages, Interactive Systems. Design, Specification, and Verification, 10th International Workshop, DSV-IS 2003, Revised Papers (Springer-Verlag, Funchal, Madeira Island, Portugal, 2003) 377–391

[41] H. Paulheim, Ontologies for User Interface Integration, Proceedings of the 8th International Semantic Web Conference, ISWC '09 (Springer-Verlag, Berlin, Heidelberg, 2009) 973–981

[42] T. Tudorache, N.F. Noy, S.M. Falconer, M.A. Musen, A knowledge base driven user interface for collaborative

ontology development, Proceedings of the 16th international conference on Intelligent user interfaces, IUI '11 (ACM, New York, USA, 2011) 411–414

[43] G. Kösters, H.W. Six, J. Voss, Combined Analysis of User Interface and Domain Requirements, Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96) (IEEE Computer Society, Washington DC, USA, 1996)

[44] K. Tilly, Z. Porkoláb, Semantic User Interfaces, IJEIS 6(1), 29–43, 2010

[45] I. Luković, S. Ristić, A. Popović, P. Mogin, An approach to the platform independent specification of a business application, Central European Conference on Information and Intelligent Systems (University of Zagreb, Faculty of Organization and Informatics Varaždin, Croatia, Hrvatska, 2011) 449–456

[46] A. Kelshchev, V. Gribova, From an Ontology-Oriented Approach Conception to User Interface Development, ITHEA, International Journal ITA (Institue of Mathematics and Informatics, Bulgarian Academy of Sciences) (Institute of Information Theories and Applications FOI ITHEA, Sofia, Bulgaria, 2003)

[47] B. Liu, H. Chen, W. He, Deriving User Interface from Ontologies: A Model-Based Approach, Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '05 (IEEE Computer Society, Washington DC, USA, 2005) 254–259

[48] J. Kollár, S. Chodarev, E. Pietriková, Ľ. Wassermann, D. Hrnčič, M. Mernik, Reverse Language Engineering: Program Analysis and Language Inference, Informatics'2011 : proceedings of the Eleventh International Conference on Informatics (TU, Košice, 2011) 109–114

[49] O. Vasilecas, D. Kalibatiene, G. Guizzardi, Towards a Formal Method for the Transformation of Ontology Axioms to Application Domain Rules, Inform. Technol. Control 38(4), 271–282, 2009

[50] D. Hrnčič, M. Mernik, B.R. Bryant, Embedding DSLs into GPLs: A Grammatical Inference Approach, Inform. Technol. Control 40(4), 307–315, 2011

[51] M. Sabo, J. Porubän, D. Lakatoš, M. Kreutzová, Computer Language Notation Specification through Program Examples, FedCSIS : Proceedings of the Federated Conference on Computer Science and Information Systems (IEEE Computer Society Press, Szczecin, Poland, 2011) 895–898