Design of Domain-Specific Software Systems with Parametric Code Templates

Kostyantyn Yermashov, Karsten Wolke, and Karl Hayo Siemsen

Abstract—Domain-specific languages describe specific solutions to problems in the application domain. Traditionally they form a solution composing black-box abstractions together. This, usually, involves non-deep transformations over the target model. In this paper we argue that it is potentially powerful to operate with grey-box abstractions to build a domain-specific software system. We present parametric code templates as grey-box abstractions and conceptual tools to encapsulate and manipulate these templates. Manipulations introduce template’s merging routines and can be defined in a generic way. This involves reasoning mechanisms at the code templates level. We introduce the concept of Neurath Modelling Language (NML) that operates with parametric code templates and specifies a visualisation mapping mechanism for target models. Finally we provide an example of calculating a domain-specific software system with predefined NML elements.

Keywords—software design, code templates, domain-specific languages, modelling languages, generic tools

I. INTRODUCTION

The role and importance of software systems in industry is crucial. The way from problem definition to software solution typically includes coherent phases of requirements specification, design, construction, testing and maintenance. Generally, the quality of the resulted software depends on how each phase is gone through.

Design is one of most challenging phases. During the software design process a solution that meets predefined requirements is produced. Modern implementations require more and more work to produce huge amount of source code. A designer regularly operates by architecturally the same or similar source code structures. It is true especially if the developer works within a strictly defined application domain. Various techniques have been suggested in the literature to domain-specific development. Benefits of domain-specific development are amplified when using a visual notation instead of textual one. Many problem domains can be modelled more successfully by experts using visual notations as they often represent problems more intuitively. Traditional implementations of domain-specific (visual) languages (DSLs) are based on a composition, where simpler objects are combined into more complex ones. For example a DSL, based on components composition allows composition of domain-specific systems with predefined pieces. Often the developer needs to control, configure and modify features distributed over different components or classes within the target software system. For example, he may want to apply the observer mechanism feature over a group of components when object components will notify subject components under certain circumstances. The routine of this feature implementation and automatic code generation may involve a sequence of activities such as code templates encapsulation, annotation, reasoning and merging.

We have defined Neurath Modelling Language (NML) - a method to visually design domain-specific software systems by means of templates merging, their configuration and transformation. Elements of NML are parametric code templates, referred to as Neurath Modelling Components (NMCs), and operations. Operations represent rules to manipulate NMCs. The rationale of the NML is not only to ease the design process for the end-user or domain-expert, but also to give them more ownership and control over the design process.

This article introduces the concept of NML. We will concentrate on the specification of language elements and their application to build a target software system. The visualization mapping mechanism is out of scope for this article.

The next section will give an overview to domain-specific (visual) languages, and black-box and gray-box abstractions. After this we describe the concept of Neurath Modelling Language. The article ends with an example and conclusion.

II. PRELIMINARY

A. Domain Specific Languages

Domain-specific Languages (DSL) or little languages are those that are tailored to a particular problem domain. Through the appropriate use of notations and abstractions they provide the expressive power to better describe specific solutions to problems in that domain [1]. Advantages of DSL are expression at an appropriate level of abstraction, employment of the concepts familiar to practitioners and better validation and optimisation at the domain level. DSL examples are Graphviz, HTML (HyperText Markup Language), SQL (Structured Query Language). DSL can be seen as composition of DSL components designed by domain expert. According to [2], DSL components describe properties of a language, e.g. parts of the lexical or syntactical structure, scope rules, typing, or the mapping to a target language. Visual notations and abstractions for DSLs are more appropriate to model systems. When using visual notations instead of textual ones for DSLs we speak about Domain-specific Visual Languages (DSVL) [3]. The quality of visual representations and the level of
of how they are accessible to the human intuition depend on information visualisation technique used. Information visualisation is the visual presentation of abstract information spaces and structures to facilitate their rapid assimilation and understanding [4]. The complexity of information can be reduced by means of information visualisation methodologies and concepts [5]–[7].

B. Black-box and Grey-box abstractions

Software systems are designed by composing existing predefined elements. A developer uses design entities of different level of abstraction, like for example, classes, components and functions. Classes and functions do represent relatively atomic level of abstraction. Quite successfully visual languages based on the component concept were specified. There are many definitions of components. According to Souza and Wills in [8] a component is a reusable part of software, which is independently developed and can be combined with other components to build larger units. Components fall more in the black-box category of abstraction [9]. This category concentrates more on the implementation of the problem. Grey-box abstractions, like patterns or source code templates, do represent part of the implementation. However they contain not yet refined abstract parts. The ongoing research on DSVLs that use grey-box abstractions is quite extensive. In this article we discuss the approach that uses code templates to build domain-specific software systems.

We integrate concepts of parametric templates, operations and domain-specific (visual) languages into the Neurath Modelling Language concepts. This is introduced in the following section.

III. NEURATH MODELLING LANGUAGE

The aim of the Neurath Modelling Language (NML) is to visually build domain-specific software using parametric, annotated programming code templates. Generally, the idea is to encapsulate these templates by hierarchically organising them in a tree or graph structure, describing them with ontologies, and providing visualisation and mapping mechanisms.

NML is a visual language, elements of which are Neurath Modelling Components (NMC) and operations. Different sets of NMCs and operations may form families of domain-specific languages.

Figure 1 depicts a simple example of NML from the developer point of view. It shows two states of the programming code written in Java and possible visual semantic interpretation. Elements defined for NML to model this domain bring the system from the state "A" to the state "B". The initial state "A" represents a Sensor entity defined as a class which is simplified to a public class with constructor. Visual representation of the code template’s meaning is the rectangle with the letter "S". After applying the operation "set new property" supplied with input parameters the target system comes to the state "B". At this state the Sensor entity implements a new feature - the ability to hold, initialize and access property temperature of the type double. At this state, the visual representation of code template’s meaning is the rectangle with the letter "S", the thermometer isotype, the default value field and the measurement notation. The next section gives more details to NMCs and operations.

A. Neurath Modelling Components

Neurath Modelling Component (NMC) is an encapsulated - hierarchically organized, structured and annotated program code of some language. NMCs represent parametric code templates supplied with meta-information which specify system and design requirements. These requirements play a crucial role in the system composition and visualisation mapping during the design (modelling) process. It is up to the domain-expert to specify those requirements. Figure 2 depicts the basic architecture of NMC.

Organisation and structuring the programming code assumes splitting the code into the smallest primitive constructs, classify them, annotate and then provide primitive operations over them. We take the Abstract Syntax Language Tree (ASLT) framework, defined by Wolke in [10] and [11], as a convenient way to encapsulate programming code constructions. ASLT is characterized as a generic concept. It works with tree or graph nodes as manipulation targets and provides primitive operations to manipulate a tree. The ASLT framework provides forward/reverse engineering tools as well as, meta-information processing tool (MIPT) to generically define manipulations over tree nodes. ASLT may be suited to encapsulate code of different languages. At the moment we have defined a complete taxonomy to encapsulate Java (version 1.4) constructs. Figure 3 depicts the basic architecture of the ASLT Framework. The ASLT API gives a lot of freedom to modify the source code. Tools can use that API to analyze source code, mark it, generate or manipulate it in other ways.
NMC uses the power of ASLT to bring entities and manipulations to the code template level. Figure 4 shows an example of NMC described in form of the ASLT structure for the following source code example written in Java:

```java
package test;
public class A{
    public void send(B b, String input){
        b.receive(input);
    }
}
```

Each node of the tree represents a primitive construction of the programming logic this ASLT tree holds. In this article we propose a notation for the ASLT types, see Table I. Nodes denoted as $\alpha$, $\beta$ and $\chi$ are meta-information nodes and represent an ontology. In software engineering an ontology defines classes of objects and relations among them and enables their automated reasoning and inference [12]. Noy and McGuinness in [13] specified basic reasons to use ontology in software engineering. Each node in ASLT, if it is not a meta-information node, represents a correspondent construct of Java and it is restricted to the number of children and their types. Parent-child relations represent hierarchy in the source code. For example, the class node $Cl$ can contain only one class body node $CB$ and this node can contain multiple method signature nodes $MS$.

Certain nodes of the example are grouped and form a template. Templates are potential targets for manipulation routines, for example extraction, injection and refactoring operations. Table II explains all groupings from the example. For instance, $\alpha$ and $\beta$ groupings represent meta-information nodes which form an ontology. Grouping denoted as $\alpha$ unifies the source code template `class A { <body> }'. Groupings $\alpha$ and $\beta$ are grouped in one group denoted as $A$.

### B. Operations

The design process based on NMCs is expressed by the transformation steps. In this article we use the term transformation to denote an act of changing one program into another. When we speak about transformation we also mean, for example, program synthesis, refinement and calculation. An operation in NML is a rule of how target NMC(s) will be transformed. Such operations are characterized by the following:

1. Formal specification
   a) Preconditions - requirements for a target entity (any concern encapsulated via NMC)
   b) Transformation rule - a formal description of the transformation of a concern
   c) Postconditions - requirements to prove semantic integrity of the model after a transformation
The domain defines three types of components - neutral, event listener and mixed. Neutral components do not pose together by connecting components with the observation mechanism defined by Gamma in [14] as observer pattern. The domain is referred to as “event driven communication” domain.

Within the domain, a software system can be composed of neutral components with implemented event-fire mechanism. Event listener components are transformed neutral components with implemented event-listen mechanism. Mixed components are transformed neutral components with implemented event-listen mechanism. Within the domain it is possible to model a static system consisting of multiple components (event sources, listeners or mixed) which can be potentially connected with events.

Figure 6 shows an example of a domain-specific system. It depicts an event source Generator and event listener Controller components. Connection between them with the directed arrow means establishing an observer mechanism for potential communication between these components. Signature E denotes an event object which is transmitted on an event. The target programming language is Java. First we specify twelve parametric templates encapsulated with NMCs, which are used to design a domain-specific software system. Then we define operations involved to manipulate the design. Finally, the predefined templates and operations are used to build a part of domain-specific software system. The following NMCs are defined:

1) “Class” NMC is referred to as NMC_class(name). It represents the source code template for a class entity in Java programming language. The parameter name during the instantiation of the NMC is set to the name of the class. First big letter of a parameter means demand to have the first letter of the value the parameter holds in upper case, to make generated code meet programming language agreements. Using terms mentioned above, a class is initially a neutral entity. The following code template is encapsulated:

```java
public class <Name>{}
public <Name>({});
```

2) “Event source interface” NMC is referred to as NMC_src(event type). The component represents an interface that an event source should implement. Specified methods represent register/remove listener and fire event mechanisms. The parameter event type holds the value that denotes an event type. The following source code template is encapsulated:

```java
public interface <Event type>Source{
    public void add<Event type>Listener(<Event type>Listener listener);
    public void remove<Event type>Listener(<Event type>Listener listener);
    public void fire<Event type>(<Event type> event);
}
```

3) “Add method” NMC is referred to as NMC_add(event type). The component represents the addXxxListener method to register new listeners in registry listenerList. The following source code template is encapsulated:

```java
public <Event type>Source{
    public void addXxxListener(<Event type>Listener listener);
}
```

Fig. 6. A model sample for the “event driven communication” domain

send nor receive events as they do not apply any event-fire-listen mechanism. Event source components are transformed neutral components with implemented event-fire mechanism. Event listener components are transformed neutral components with implemented event-listen mechanism. Mixed components are transformed neutral components with implemented event-listen mechanism.
4) "Remove method" NMC is referred to as NMC<sub>rem</sub>(event type). The component represents the removeXxxListener method to remove listeners. The following source code template is encapsulated:

```java
public void remove<Event name>Listener(
    <Event name>:Listener listener|
    listenerList.remove(<Event name>Listener.class,
    listener);
}
```

5) "Fire method" NMC is referred to as NMC<sub>fire</sub>(event type). The component represents the fireXxx method to notify all registered listeners. The following source code template is encapsulated:

```java
public void fire<Event type>(<Event type> event){
    Object[] listeners = listenerList.getListeners();
    for(int i=0; i<listeners.length; i+=2) {
        if(listeners[i]==<Listener name>Listener.class){
            ((<Listener name>Listener)listeners[i+1]).
            <event type>Occurred(<Event type> event);
        }
    }
}
```

6) "Listener interface" NMC is referred to as NMC<sub>list</sub>(event type). The component represents the variable listenerList declaration and initialisation, that holds all registered listeners. The following source code template is encapsulated:

```java
protected EventListenerList
    listenerList = new EventListenerList();
```

7) "Listener registry" NMC is referred to as NMC<sub>reg</sub>. The component represents the variable listenerList declaration and initialisation, that holds all registered listeners. The following source code template is encapsulated:

```java
public interface <Event type>Listener{
    public <Event type>(Object source){
        super(source);}
}
```

8) "Event type" NMC is referred to as NMC<sub>evt</sub>(event type). The component holds the class template which defines an event type. The event object holds a reference to the event source. The following source code template is encapsulated:

```java
public class <Event type> extends EventObject{
    public <Event type>(Object source){
        super(source);
    }
}
```

We will construct the model using parametric code templates instantiations, domain-specific operations "source injection" and "listener injection" - denoted as @src and @list accordingly - and several atomic ones. This will be shown on each transformation iteration.

1) Initialisation of neutral elements with names Controller and Generator:

\[ NMC_{class1} = NMC_{class}(Controller) \]  (1)

\[ NMC_{class2} = NMC_{class}(Generator) \]  (2)

\[ System = \oplus^{aslt}(root, NMC_{class1}) \]  (3)

\[ System_1 = \oplus^{aslt}(System, NMC_{class2}) \]  (4)

The atomic operator \( \oplus^{aslt} \) attaches templates as sub-trees into the ASLT according to requirements specified for node’s children, so that the syntax of generated java programming code is sound. For example, in (3) the NMC<sub>class1</sub> is attached to the root that represents the root node of the ASLT.

2) Transformation of neutral component NMC<sub>class1</sub> into the event listener component. This is done with the help of "listener injection" \( \oplus^{lst} \) operation:

\[ \oplus^{lst}(a, b) = \oplus^{i11}(a, b) \land \oplus^{i12}(a, b) \]  (5)

where \( \land \) is a separator between operations, and \( \oplus^{i11} \) and \( \oplus^{i12} \) are defined as follows:

\[ \oplus^{i11}(a, b)=\oplus^{aslt}(fnl(b, nmc<sub>imp</sub>), fnl(a, nmc<sub>n</sub>)) \]  (6)

\( \oplus^{i11} \) extends an interface list of target b by the interface a. Operation \( fnl \) is an atomic one, it searches sub-tree defined by the constant. The constant pattern \( nmc<sub>n</sub> \) specifies a pattern to find the interface name match, and \( nmc<sub>imp</sub> \) - to find an interface implementation list match.

\[ \oplus^{i12}(a, b)=\oplus^{aslt}(fnl(b, nmc<sub>n</sub>), rvl(a, nmc<sub>n</sub>)) \]  (7)

\( \oplus^{i12} \) reveals methods from the interface a to be implemented in b and generates correspondent method’s declarations within b. The operation \( rvl \) in (7) is an atomic one, it reveals from a sub-trees defined by the constant. In this case it is the constant \( nmc<sub>n</sub> \) that characterizes all event handler method declarations. The constant pattern \( nmc<sub>n</sub> \) specifies a pattern to find a place where methods are declared.

Now in (5) we set the NMC<sub>class1</sub> instead of b and the formula (8) instead of a, see (9).

\[ NMC_{list1} = NMC_{list}(AccEvt) \]  (8)

\[ NMC_{list1} = \oplus^{lst}(NMC_{list1}, NMC_{class1}) \]  (9)

Additionally, in (10) the system is extended with newly generated class NMC<sub>evt</sub> = NMC<sub>evt</sub>(AccEvt) describing events:

\[ System_3=\oplus^{aslt}(System_2, NMC_{evt}) \]  (10)

The word AccEvt is the name of an event type required by the "Event type" NMC.

3) Transformation of the neutral component NMC<sub>class2</sub> into the event source component. This is done with help of "source injection" \( \oplus^{src} \) operation:

\[ \oplus^{src}(a, b, evt) = \oplus^{i11}(a, b) \land \oplus^{m1}(b, evt) \]  (11)

where b is a target class to implement the interface a, \( \oplus^{src} \) is a type of the event and the operation \( \oplus^{m1} \) is defined as follows:

\[ \oplus^{m1}(b, evt)=\oplus^{reg}(b) \land \oplus^{add}(b, evt) \land \oplus^{rem}(b, evt) \land \oplus^{fire}(b, evt) \]  (12)
This domain-specific operation transforms the large code entity so that meets requirements defined for the event source. It produces an additional event source interface, then a declaration of registry to hold registered event listeners, and implementations of methods add, rem and fire to register, remove and throw an event consequently. $\oplus^{reg}$, $\oplus^{add}$, $\oplus^{rem}$ and $\oplus^{fire}$ are defined as follows:

$$\oplus^{reg}(b) = \oplus^{aslt}(\text{find}(b, nmc_{decl}), NMC_{reg})$$ (13)

$$\oplus^{add}(b, e) = \oplus^{aslt}(\text{find}(b, nmc_{m}), NMC_{add}(e))$$ (14)

$$\oplus^{rem}(b, e) = \oplus^{aslt}(\text{find}(b, nmc_{m}), NMC_{rem}(e))$$ (15)

$$\oplus^{fire}(b, e) = \oplus^{aslt}(\text{find}(b, nmc_{m}), NMC_{fire}(e))$$ (16)

The $\oplus^{reg}$ operation inserts an event registry definition to the position defined by the constant nmc$_{decl}$. The constant specifies a pattern to find a place within the ASLT where global variables are declared. The $\oplus^{add}$, $\oplus^{rem}$ and $\oplus^{fire}$ operations insert methods implementations to the position defined by the constant nmc$_{m}$. The constant specifies a pattern to find a place within the programming code encapsulation where methods are defined. The parameter $e$ holds a value that denotes an event type, specified above. Now in the formula (11) we replace $a$ and $b$ with NMC$_{isrc}$ defined in (17) and NMC$_{class2}$ defined in (2) respectively, see (18).

$$NMC_{isrc} = \oplus^{fire}(NMC_{isrc}, NMC_{class2}, AccEvt)$$ (17)

$$NMC_{isrc} = \oplus^{fire}(NMC_{isrc}, NMC_{class2}, AccEvt)$$ (18)

At this point the System$_{3}$ represents a part of implementation of the domain software system. It describes the implementation of static system within the “event driven communication” domain. System$_{3}$ is characterized by the event listener and the event source entity, which may be dynamically connected in order to communicate with simple events.

IV. CONCLUSION

We highlighted in this paper the concept of modelling language for synthesis of domain-specific software systems with parametric code templates. Encapsulation of such templates, which structures, deeply classifies, and annotates them, together with generic reasoning and manipulation mechanisms results an effective design of domain-specific software systems. The example showed steps to build such a system with predefined elements.

Our future work will concentrate on the enhancement of the provided tool-support for specification and analysis. Additionally, we aim to concentrate on visual mapping strategies for the NML as well as optimisation of calculation of domain-specific software systems.

REFERENCES


[14] E. Gamma, R. Helm, and E. Johnson, Ralph, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Professional Computing Series, 1994, iSBN: 0201633612.