International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:10, 2008

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.

Kostyantyn Yermashov and Karsten Wolke are with Laboratory of Parallel Processes, University of Applied Sciences, Constantiaplatz 4, DE-26721, Emden, Germany, and Software Technology Research Laboratory, De Montfort University, Leicester, LE1 9BH, United Kingdom (e-mail: konstant,wolke@ossi.fho-emden.de).

Karl Hayo Siemsen is with Laboratory of Parallel Processes, University of Applied Sciences, Constantiaplatz 4, DE-26721, Emden, Germany, (e-mail: siemsen@ossi.fho-emden.de).

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 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 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 ”insert 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

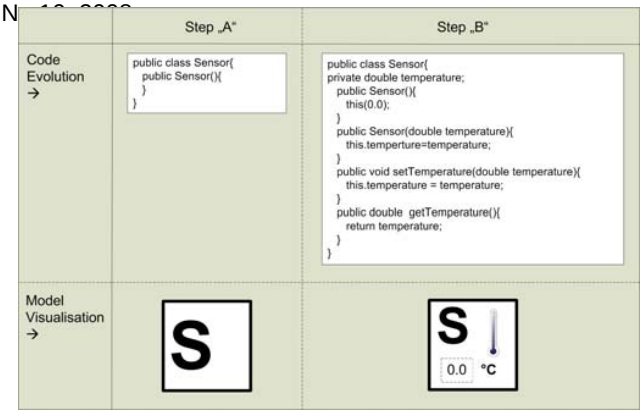

Fig. 1. Example of a program transformation

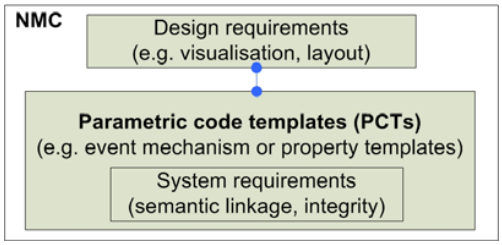

Fig. 2. Architecture of NMC

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.

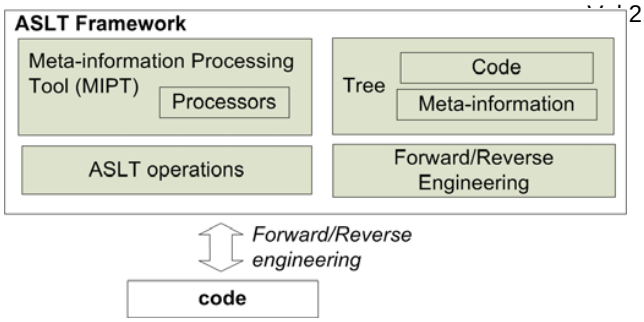

Fig. 3. ASLT Framework

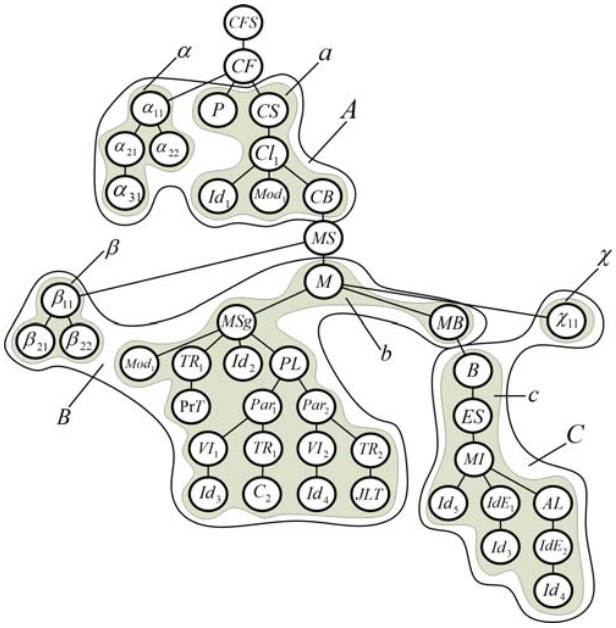

Fig. 4. NMC example

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:

```
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 α, β and χ 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*.

TABLE I
NOTATION FOR PROGRAMMING LANGUAGE CONSTRUCTS

Sign.	Description	Sign.	Description
CFS	Set of source code files	*CF*	Source code file
P	Package	*CS*	Set of classes
Cl	Class	*Id*	Identifier
Mod	Modifier	*CB*	Body of a class
MS	Set of methods	*M*	Method
MSg	Method signature	*MB*	Method body
TR	Reference to a type	*PL*	List of parameters
PrT	Primitive type	*Par*	Parameter
VI	Identifier of the variable	*JLT*	Library type
B	Block	*ES*	Expression statement
MI	Method invocation	*IdE*	Identifier expression
AL	Argument list		

TABLE II
GROUPINGS OF ASLT NODES

Groupings (templates)	Information grouped
α	Ontology related to the grouping a
a	`class A { <body> }`
A	relates a and α together
β	Ontology related to the grouping b
b	`public void send(B b, String input) { <body> }`
B	relates b and β together
χ	Ontology related to the grouping c
c	`b.receive(input)`
C	relates c and χ together

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, α and β groupings represent meta-information nodes which form an ontology. Grouping denoted as a unifies the source code template `class A { <body> }`. Groupings α and a 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

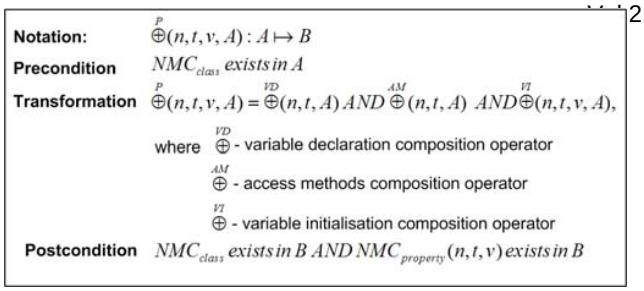

Fig. 5. Specification of the "property composition" operation

2) Code templates to be used when applying an operation
3) Other resources, for example resources related to visualisation routines

Figure 5 shows the conceptual specification example of the operation "property composition". The operation transforms the A template into the B template so that the resulting B implements a property described by the input parameters n, t and v, where n is a property name, t is a property type and v is an initial value. The transformation is defined as a composition of operators $\oplus^{VD}$, $\oplus^{AM}$, $\oplus^{VI}$. Similar operations with more concrete definition of transformation rules can be found in the section III-C. With postconditions the semantic integrity can be proved.

We separate between domain-specific, molecular and atomic operations. Atomic operations are composed of primitive ASLT operations and represent manipulations with templates, for example find template match in a tree. We developed and unified atomic operations in the ASLT atomic manipulation library. For example, the following operations are provided:

- Find first/all node(s) by type and/or value
- Find first/all sub-tree(s) by tree-pattern
- Find all sub-tree roots from the given hierarchy level
- Insert/replace/extract a sub-tree
- Extract linkage defined by meta-information nodes
- Inject values defined by some tree-pattern into a tree

A molecular operation is any operation composed of atomic ones. For example, the operation "find method called *send* and extract it". Domain-specific operations represent transformations which implement domain features defined or directly used by the domain expert. For example, the operation "connect tubes with pipes" belongs to the group of domain-specific operations.

The next section gives an example of NMCs and operations which form a subset of NML to model the "event driven communication" domain.

C. Example

In this section we give a simple example of components and operations of NML to model a system within a certain domain. The domain is referred to as "event driven communication" domain. Within the domain a software system can be composed together by connecting components with the observation mechanism defined by Gamma in [14] as observer pattern. The domain defines three types of components - neutral, event source, event listener and mixed. Neutral components do not 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-fire-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.

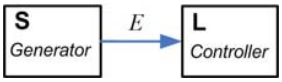

Fig. 6. A model sample for the "event driven communication" domain

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:

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

2) *"Event source interface" NMC* is referred to as $NMC_{isrc}\,(event\ type)$. 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:

```
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:

```
public void add<Event type>Listener(
  <Event type>Listener listener){
  listenerList.add(<Event name>Listener.class,
  listener);
}
```

4) *"Remove method" NMC* is referred to as NMC_{rem} (*event type*). The component represents the *removeXxxListener* method to remove listeners. The following source code template is encapsulated:

```
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_{fire} (*event type*). The component represents the *fireXxx* method to notify all registered listeners. The following source code template is encapsulated:

```
public void fire<Event type>(<Event type> event){
  Object[] listeners;
  listeners = listenerList.getListenerList();
  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);
   }
  }
}
```

6) *"Listener interface" NMC* is referred to as NMC_{ilst} (*event type*). The component represents the event listener interface. All listeners of an event defined by the parameter *event type* implement this interface. The specified method is called for each listener when an event is fired. The following source code template is encapsulated:

```
public interface <Event type>Listener
  extends java.util.EventListener{
  public void <event type>Occurred(<Event type>
  event);
}
```

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

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

8) *"Event type" NMC* is referred to as NMC_{event} (*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:

```
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 $\oplus^{src}$ and $\oplus^{lst}$ accordingly - and several atomic ones. This will be shown on each transformation iteration.

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

$$NMC_{class_1} = NMC_{class}\,(Controller) \qquad (1)$$

$$NMC_{class_2} = NMC_{class}\,(Generator) \qquad (2)$$

$$System = \oplus^{aslt}\,(root, NMC_{class_1}) \qquad (3)$$

$$System_1 = \oplus^{aslt}\,(System, NMC_{class_2}) \qquad (4)$$

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_{class_1} is attached to the *root* that represents the root node of the ASLT.

2) Transformation of neutral component NMC_{class_1} into the event listener component. This is done with the help of "listener injection" $\oplus^{lst}$ operation:

$$\oplus^{lst}\,(a,b) = \oplus^{ii_1}\,(a,b)\ \wedge\ \oplus^{ii_2}\,(a,b) \qquad (5)$$

where $\wedge$ is a separator between operations, and $\oplus^{ii_1}$ and $\oplus^{ii_2}$ are defined as follows:

$$\oplus^{ii_1}(a,b) = \oplus^{aslt}(fnd(b, nmc_{imp}), fnd(a, nmc_{in})) \qquad (6)$$

$\oplus^{ii_1}$ extends an interface list of target b by the interface a. Operation *fnd* is an atomic one, it searches sub-tree defined by the constant. The constant pattern nmc_{in} specifies a pattern to find the interface name match, and nmc_{imp} - to find an interface implementation list match.

$$\oplus^{ii_2}\,(a,b) = \oplus^{aslt}\,(fnd(b, nmc_m), rvl(a, nmc_h)) \qquad (7)$$

$\oplus^{ii_2}$ 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_h that characterizes all event handler method declarations. The constant pattern nmc_m specifies a pattern to find a place where methods are declared.

Now in (5) we set the NMC_{class_1} instead of b and the formula (8) instead of a, see (9).

$$NMC_{ilst_1} = NMC_{ilst}(AccEvt) \qquad (8)$$

$$NMC_{lst_1} = \oplus^{lst}(NMC_{ilst_1}, NMC_{class_1}) \qquad (9)$$

Additionally, in (10) the system is extended with newly generated class NMC_{evt_1}=$NMC_{evt}(AccEvt)$ describing events:

$$System_3 = \oplus^{aslt}(\,System_2, NMC_{evt_1}\,) \qquad (10)$$

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

3) Transformation of the neutral component NMC_{class_2} into the event source component. This is done with help of "source injection" $\oplus^{src}$ operation:

$$\oplus^{src}\,(a,b,evt) = \oplus^{ii_1}\,(a,b) \wedge \oplus^{mi_1}(b,evt) \qquad (11)$$

where b is a target class to implement the interface a, evt is a type of the event and the operation $\oplus^{mi_1}(b,evt)$ is defined as follows:

$$\oplus^{mi_1}\,(b,evt) = \oplus^{reg}\,(b) \wedge \oplus^{add}(b,evt) \wedge \oplus^{rem}(b,evt) \wedge \oplus^{fire}(b,evt) \qquad (12)$$

This domain-specific operation transforms the target code entity so, that is 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 correspondently. $\oplus^{reg}$, $\oplus^{add}$, $\oplus^{rem}$ and $\oplus^{fire}$ are defined as follows:

$$\oplus^{reg}(b) = \oplus^{aslt}(\mathit{fnd}(b, \mathit{nmc}_{decl}), \mathit{NMC}_{reg}) \quad (13)$$

$$\oplus^{add}(b, e) = \oplus^{aslt}(\mathit{fnd}(b, \mathit{nmc}_{m}), \mathit{NMC}_{add}(e)) \quad (14)$$

$$\oplus^{rem}(b, e) = \oplus^{aslt}(\mathit{fnd}(b, \mathit{nmc}_{m}), \mathit{NMC}_{rem}(e)) \quad (15)$$

$$\oplus^{fire}(b, e) = \oplus^{aslt}(\mathit{fnd}(b, \mathit{nmc}_{m}), \mathit{NMC}_{fire}(e)) \quad (16)$$

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_1} defined in (17) and NMC_{class_2} defined in (2) respectively, see (18).

$$\mathit{NMC}_{isrc_1} = \mathit{NMC}_{isrc}(AccEvt) \quad (17)$$

$$\mathit{NMC}_{src_1} = \oplus^{src}(\mathit{NMC}_{isrc_1}, \mathit{NMC}_{class_2}, AccEvt) \quad (18)$$

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

[1] D. J.M. Taylor and L.J. Mazlack, *Domain-Specific Ontology Merging for the Semantic Web*, NAFIPS 2005 Annual Meeting of the North American Fuzzy Information Processing Society, 2005.

[2] P. Pfahler and U. Kastens, *Configuring Component-Based Specifications for Domain-Specific Languages*, Proceedings of the 34th Hawaii International Conference on System Sciences, 2001.

[3] R. Esser and J. W. Janneck, "A framework for defining domain-specific visual languages," *In Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA-2001*, 2001.

[4] *Information Visualisation. Tutorial Notes*, http://www.iicm.edu/-ivis/-ivis.pdf, 1998.

[5] P. Irani, M. Tingley, and C. Ware, "Using Perceptual Syntax to Enhance Semantic Content in Diagrams," *IEEE Computer Graphics and Applications*, vol. Vol. 21, No. 5, pp. pp. 76–84, 2001.

[6] P. Honeywill, "A comparison between maya hieroglyphs and computer icons," *AI & Society archive*, vol. Vol. 14, Issue 3-4, pp. pp.395–410, 2000, iSSN: 09515666.

[7] O. Neurath, "Visual education: A new language," *Survey Graphic*, vol. Vol. 26, No. 1, 1937.

[8] D.D. Souza and A. C. Wills, *Objects, Components and Frameworks: The Catalysis Approach*. Addison-Wesley, 1998.

[9] D. Alur, J.Crupi, and D.Malks, "Core J2EE Patterns, Best Practices and Design Illusions," *Proceedings of the Twentieth Information Systems Research Seminar in Scandinavia*, 1997.

[10] K. Wolke, *ASLT Framework*, LaborPP, University of Applied Sciences, Emden (Germany) and STRL, De Montfort University Leicester (UK), http://www.karsten-wolke.de/public/aslt/ASLT_1.1.rar, 2006.

[11] K. Wolke, *Meta Information in ASLTs*, LaborPP, University of Applied Sciences, Emden (Germany) and STRL, De Montfort University Leicester (UK), http://www.karsten-wolke.de/public/aslt-/ASLTMetaData.pdf, 2006.

[12] M. Solanki, *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Services (PhD Thesis)*, Software Technology Research Laboratory, De Montfort University, 2005.

[13] N. F. Noy and D. L. McGuinness, *Ontology Development 101: A Guide to Creating Your First Ontology*, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, 2001.

[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.