Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2011
Volume 47

The Definition of Framework for Automated Creation of Graph Visualization Systems

Vitaly Zabiniako[1], Pavel Rusakov[2], [1-2]*Riga Technical Univesity*

***Abstract* – In this paper the authors perform an analysis of automated creation of graph visualization systems (GVS). The complete process, starting with definition of functional requirements for GVS and finishing with evaluation of resulting system, is described. The overview of each construction step is provided. The set of metric types is developed for evaluation of resulting GVS. Features of existing solutions are compared with proposed framework. Conclusions are made and further researches are defined.**

Keywords – automation, framework, graph, visualization.

I. Introduction

The concept of automated creation of software systems became a popular research topic among IT community nowadays. Latest trends in this domain involve usage of modern tools and approaches – software factoring, MDA (model driven architecture) [1], DSL (domain specific languages) [2].

There is a little doubt that full-fledged automatic creation of software, given such efforts, will emerge in relatively near future. The main problem being analyzed is a need to explore these possibilities further while advancing existing semi-automated methods. This paper introduces the appropriate theoretical background in the domain of information (in particular – graph) visualization and proposes the partial solution of such problem. We believe that this domain is too sensitive to rely on fully automatic software generation.

Processing of input data from the user that defines a set of functional and non-functional requirements for the GVS is the one key-point of the proposed framework. The second aspect is the development and maintaining of the repository that holds "building bricks" with desired functionality. This repository has to manage the relationships between these and functional requirements and allow building seamlessly, verifying, and evaluating resulting GVS. Such modular approach allows common benefits of software factoring – ability to reuse GVS components along with support of design templates that allows to build and customize a visualization solution.

The goal of this research is to support further development of tools and methods for automated software generation and creation of GVS systems in particular.

To reach this goal, the following five tasks are defined: 1) to provide general overview of proposed framework and outline cross-references between its individual steps, user data workflow and intermediate results; 2) to describe each framework element in details; 3) to present a set of metric types for evaluation of GVS and its conformance to the intended functionality; 4) to compare presented framework with appropriate existing solutions; 5) to make a conclusion about achieved results and outline potential further work.

In the 2-nd section of this paper general description of the framework is provided along with the visual model of its components and their mutual relationships. The 3-rd section specifies each of these components in details. The 4-th section provides description of evaluation metrics required for verification and validation of the GVS performance. In the 5-th section the similarities between existing solutions and proposed framework are outlined. The 6-th section provides summary of this work and its future development.

II. General overview of proposed framework

The model of the framework is presented in Fig. 1. It has three-layer architecture in which elements within the first level correspond to GVS building processes, while the second and third levels encapsulate the inner implementation mechanisms.

The top level serves both as entry point for initial requirements specification and the final output in form of appropriate GVS. It consists of three stages – "Specification", "Assembly", and "Validation", that, in general, correspond to classical model "requirements – code – tests". This level also holds the feedback that allows iterative GVS construction, considering the refined user input after preliminary evaluation.

The second level holds a number of sets of individual components that are important parts of GVS (we will refer to these as "artifacts"). An artifact is any GVS-related entity that is self-distinguishable and unique. There are four types of artifacts in the proposed framework – "Template", "Requirement", "Module", and "Metric". Each set should be implemented as a separate repository with ability to maintain (eg. add, edit, delete, provide description) own artifacts and define relationships with other sets via cross-references.

The third level implements the core logic of the framework and holds "connectors" that allow configuring (and reconfiguring) mutual relationships between artifacts.

There are three connecting sets, two of these define connection between templates and requirements ("TR connector"), requirements, modules, and metrics ("RMM connector") while the third manages reflexive connection between modules ("M connector") – refer to the next section.

Potential framework end-user who wants to construct GVS according to his/her needs, is required to participate in two stages – "Specification", and "Validation", and optionally – in "Assembly", if he/she wishes to alter the source code of generated GVS manually. Also, there might be a manager of

the framework who configures the content and relationships for any of the framework elements in lower layers.

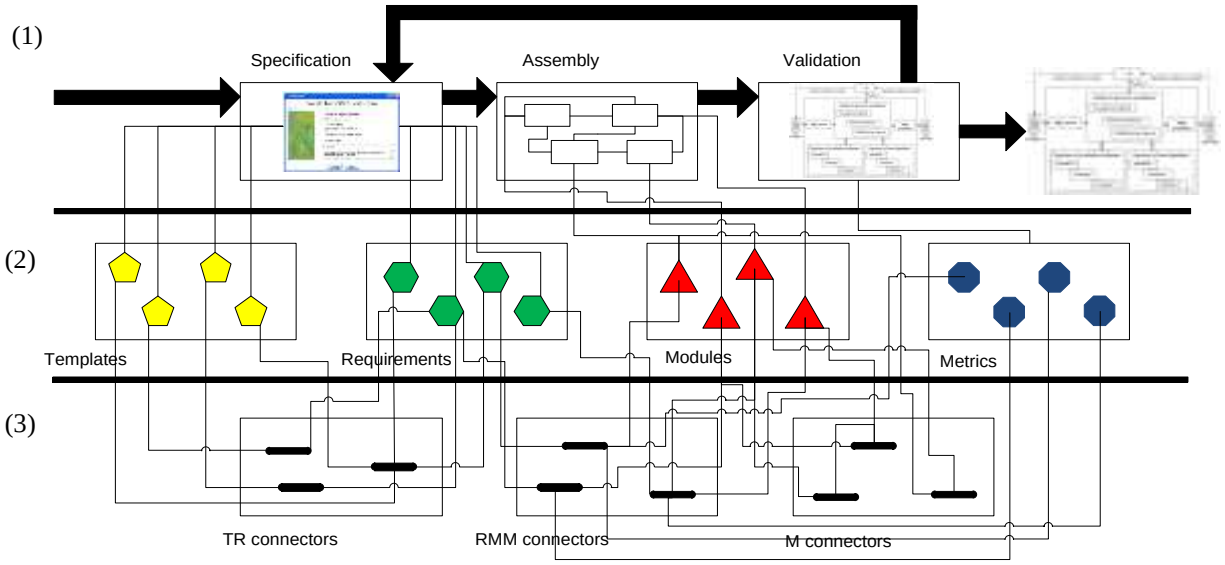

Fig. 1. General framework architecture

Although the actual implementation of the framework can be different, for further description we will rely on classic relational database model that is sufficient and appropriate, given the self-consistent nature of artifacts and mutual relations. Another mechanism we utilize in our framework is scripting languages that allow dynamic configuring of the GVS.

III. Description of Individual Framework Components

This section presents more detailed insight into each of the component of all three framework levels and provides description of automated GVS construction workflow, related assumptions and implementation specifics.

A. Components of the first layer

Stage "Specification" is the starting point that allows user to define both general and specific requirements for GVS to be built.

During this stage the user browses and marks a set of predefined requirements that are expressed in textual form. For example – "Ability to auto-place graph elements corresponding to known layouts" or "Ability to import and export produced layouts from/to known graph description languages". Each requirement can be further subdivided into more specific requirements, such as "Support of force-based layout" or "Support of GraphXML". For user's convenience and enhancement of the interaction, these requirements can be presented with additional visual content, such as appropriate images or short videos of typical effect of implementation of such requirements.

Basic user interface for this stage should be founded either on a list of requirements with a set of checkbox elements for selection or a step-by-step wizard that encapsulates definition of requirements for semantically different GVS components (i.e. layout algorithms, visual techniques, navigation, etc.) within individual steps.

In case if required GVS is intended for visualization in certain well-known domains (for example – GVS for drawing chemical molecular structures / GVS for analyzing software structures like call graphs, etc.), a higher requirement abstraction level is introduced in form of patterns. The pattern is a predefined set of requirements in a certain combination that is managed by "TR" connectors (see below in this section).

The opposite logic of level of detail is also present – even individual requirements can hold a set of properties that affect final implementation of that particular requirement. For example requirement to be able to select certain graph elements and mark this selection with distinct color must allow defining that particular color explicitly.

An example of interface prototype for this step is presented in Fig. 2.

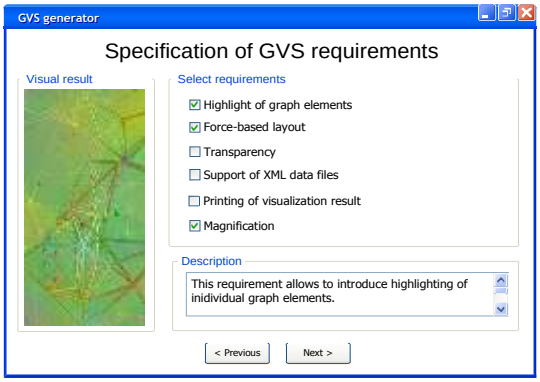

Fig. 2. Interface prototype

Stage "Assembly" is an automated process that actually performs generation of GVS, based on the results of the previous step. This part of the framework relies on multiple components of lower sub-layers and its interconnection logic. The core of this stage is a general GVS template that conforms

to the architecture that was previously proposed by the authors of this work. Its general model is presented in Fig. 3.

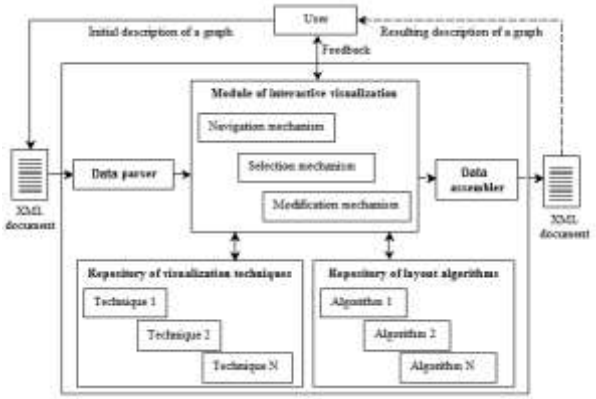

Fig. 3. GVS template architecture

It consists of five modules: data parser, module of interactive visualization, repository of layout algorithms, repository of visualization techniques, and data assembler.

Data parser is represented by a software module that performs interpretation of the document with a description of the graph provided by the user, and allows extracting necessary information about general topology of the graph and its separate elements (names of nodes and edges, colors of these, etc.).

Module of interactive visualization is the main part of the system that performs visualization of received information (by triggering selected layout algorithms from the repository of layout algorithms and visual techniques for improved comprehension from repository of visualization techniques), offers the analysis and processing abilities (for example, by using navigation, selection, and modification mechanisms if these were selected by the user during specification stage) and passes processing results to data assembler in case, if the user wants to save processed and modified data. This module includes up to three main mechanisms for interaction between user and software system itself.

Navigation mechanism allows transforming visual appearance of three-dimensional graph model by altering position of a virtual camera. *Selection mechanism* allows distinguishing and marking certain set of visible parts of a graph for further modification. *Modification mechanism* allows altering the previously selected parts of a graph topological structure and / or associated metadata.

Repository of layout algorithms holds a set of selected layout algorithms for visualization of graphs.

Repository of visualization techniques holds a set of techniques for improvement of comprehension – these can be triggered within the module of interactive visualization.

Data assembler is a software module that acts similar to data parser with the only distinction – the direction of information processing flow. Again, usage of this module is optional and is required only if user wants to save changes introduced to original graph, after data processing.

For detailed description of this template architecture, refer to [3].

Performance, content, or even each of the mentioned modules itself can be dynamically modified / replaced within the proposed framework during the assembly stage. This is possible due to dynamic configuration of the template via appropriate scripting language. We will demonstrate this concept in a simple interaction example between C++ (within GVS template) and Lua (within the proposed framework) [4].

On the side of GVS template it is necessary to set up Lua runtime components and register appropriate native functions to be called from Lua. This is done as follows:

```
/*1*/  static int GvsFnc(lua_State *L)
/*2*/  {
/*3*/    //alter graph
/*4*/    return 1;
/*5*/  }

/*6*/  int main (int argc, char *argv[])
/*7*/  {
/*8*/    L = lua_open();
/*9*/    luaL_openlibs(L);
/*10*/   lua_register(L, "FrmFnc", GvsFnc);
                              …
/*11*/   luaL_dofile(L, "framework.lua");
/*12*/   lua_call(L, 0, 0);
                              …
/*13*/   lua_close(L);
/*14*/   return 1;
/*15*/ }
```

Abstract function "GVSFnc" that performs some kind of modifications in a template (for example – changes the default color of graph nodes) is defined in code lines 1 to 5. This function is registered with Lua runtime at code line 10 within the main GVS workflow (code lines 6 to 15). Code lines 8 to 10 and 13 to 14 ensure correct Lua runtime initialization and shutdown.

On the side of the framework the control flow of GVS behavior is defined within appropriate Lua script, as follows:

```
/*1*/ function LuaFunction()
                              …
/*2*/   FrmFnc()
                              …
/*3*/ end
```

Code lines 1 and 3 define Lua script body, while the dynamic call of graph altering function within GVS template is performed in code line 2. This script can be loaded and reloaded within GVS template upon necessity.

The procedure that is necessary to identify required modules and setup these correctly, given specific requirements, is handled by RMM and M connectors (see below).

Stage "Validation" is the last entity of the first framework level. It ensures that resulting GVS meets previously defined requirements and fulfills its intended purpose according to the needs of the user.

Although we tend to provide automated or even automatic support of construction of GVS, there is no doubt that proper

automatic evaluation of the resulting GVS is still not possible, as it would require advanced AI facility much more complicated than the proposed framework itself. Additional aspect that denies fully automatic evaluation is related to the fact that the user himself/herself is the one who defines requirements with particular functionality in mind.

Instead, the validation process is partially automated based on feedback of the user and evaluation model that allow carrying out more formal analysis of the difference between intended and resulting GVS functionality.

Validation process consists of two steps: 1) a test run of resulting GVS; 2) a questionnaire with predefined questions that a user must answer. After receiving all the feedback, data about system performance is collected, calculations of conformance rating are carried out, based on a set of metrics that are related with initial requirements and chosen modules via RMM connector (see below in this section).

Considering conformance rating the user will be able either to accept functionality of the generated GVS or to repeat specification and assembly stages to modify, add or remove certain functionality. Functionality of validation stage is tightly integrated with both mentioned stages within single IDE (Integrated Development Environment) allowing seamless building process.

Specification of the proposed formal model and its individual metrics is presented in section IV.

B. Components of the second layer

A set of templates enables management of capabilities of the framework to build predefined GVS tailored for particular needs. In the simplest scenario individual template artifact is presented as a single record in the database with the only property – its name. More complex solutions, most likely, will introduce additional properties, such as textual description of the represented GVS or even preset values for individual properties of requirement artifacts. This enhanced version allows keeping "user input / framework output" ratio at a minimum – by selecting single name of the template, the user will end up triggering dynamic development of full-fledged GVS.

A set of requirements enables user to define functional behavior of target GVS. It is important to note that in this framework common scenario for the user is to choose from those requirements that already exist within the framework database rather than defining his/her own. Definition of new requirement artifacts and potentially of associated modules, metrics, and connectors should be done by the framework manager with appropriate knowledge. Same applies to any other artifact that is not present in the system. This means that the content of the framework for GVS generation should be gradually developed and expanded upon necessity by the manager of this software system.

As it was mentioned before, in the simplest scenario the requirement will be presented by its name only. Although extended description is expected to be defined too, as it is the main aspect that allows user to comprehend the meaning of intended functionality. Extended properties, such as visual image of the expected effect (if appropriate) can be introduced, although these are not crucial for successful GVS generation.

Individual properties of requirements can be defined in relational database with classical one-to-many relationship. This is especially useful in case if properties are supplemented with additional description for the convenience of the user.

A set of modules holds the source code for those functional modules that will be dynamically linked together and adjusted via scripts during the assembly stage. The simplest implementation of a single module artifact can include the record in the database with two fields – one for the name of the module and the other one for the path to a file with the source code, stored somewhere in file system of the operating system in which the framework resides. More complex model must enforce additional features, such as entry points for parameters that are supplied within requirement artifacts (refer to description above).

A good example of a module would be "Font package", "Force-based algorithm package" or "Alpha blending (transparency) package" with appropriate implementation (for detailed description of such algorithms and techniques refer to [5] and [6]).

A set of metrics is the last entity of the second framework level. It holds names, descriptions, types, and default values for individual metrics that are crucial within the validation stage. Each metric, in its essence, is a numerical value from 0 to 100 that conforms to the percentage of how fully a certain requirement is met within the resulting GVS comparing to initial expectations, according to user's opinion. Although such evaluation is highly subjective, after calculations, it provides user with a single united rating that is credible enough for this particular user to make a decision – whether to end generation process or return to the first step and repeat the generation by adjusting a set of initial requirements and its parameters (or even, if the user is trained enough – by altering the source code of the produced GVS).

C. Components of the third layer

A set of TR connectors defines one-to-many relationships between each single template artifact and multiple corresponding requirements artifacts. For example, a template of graph visualization system that deals with ATC (Air Traffic Control) routing will require modules like "colors and shapes package", "object group picking package", etc. (refer to [7] for detailed description of the requirements for such kind of a system). TR connector in this case will define cross-references between artifacts of both sets.

Here and further the formal mechanism for making of connectors is based on the following model (Fig. 4.):

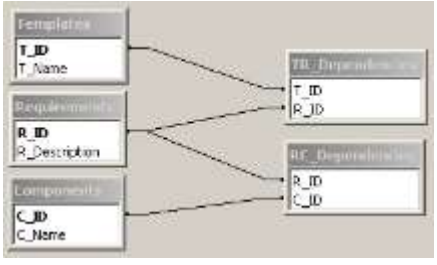

Fig. 4. Definition of relationships between artifacts

All fields in tables "Components", "Requirements", and "Templates" correspond to the previous description. Each table holds ID and names of defined artifact. The required transformation allows converting to a list of necessary GVS components from a name of an arbitrary template. In order to do so, there are two "many–to–many" relationships defined between tables. The first one defines dependencies between GVS requirements and components. The second defines a list of possible requirements for each GVS template.

The task of retrieving the list of components from the template name can be successfully accomplished with a single SQL (Structured Query Language) query:

```
/*1*/ SELECT A.C_Name
/*2*/ FROM Components AS A, Requirements AS
                    B, RC_Dependencies AS C
/*3*/ WHERE B.R_ID=C.R_ID AND A.C_ID=C.C_ID
                    AND B.R_ID In
/*4*/ (SELECT D.R_ID
/*5*/ FROM Requirements AS D, Templates AS
                    E, TR_Dependencies AS F
/*6*/ WHERE D.R_ID=F.R_ID And E.T_ID=F.T_ID
                    Ad E.T_Name="TName")
/*7*/ GROUP BY A.C_Name;
```

The inner select query (lines of code 4 to 6) allows extracting list of requirements ID for provided template name (marked bold in Fig.4) by examining related relationships in the first auxiliary table "TR_Dependencies". The outer select query (lines of code 1 to 3 and 7) takes previous result as an additional constraint while examining relationships in the second auxiliary table "RC_Dependencies" in order to retrieve the list of component names (marked bold in Fig.4). The "GROUP BY" keyword allows to get rid of duplicates in the resulting table.

A set of RMM connectors defines many-to-many relationships between requirements and modules artifacts as well as the same kind of connections between requirements and metrics. This kind of relationships differs from the previous connector type, because the same requirement may enforce inclusion of several modules / metrics and vice versa – a module / metric might be shared between requirements.

Considering that RMM connector deals with such dense cross-references, it is of great importance to provide convenient IDE for the framework manager not only for management of connectors but also for debugging and tracing of its inter-dependencies.

A set of M connectors is the last entity of the third framework level. As it might be perceived from Fig. 1., this type of connectors define mutual relationships between modules only. Such kind of additional control is necessary to ensure correct integration of different modules while avoiding potential functionality conflicts. A good example would be the case when user chooses to simultaneously include modules both for selection of graph elements with a cursor and visually enlarging (scaling up) selected objects. Visual scaling can be implemented in two ways – either with native matrix manipulation command within graphical pipeline or image post-processing activities (refer to [6]). The latter case is incompatible with default object selection mechanism which requires all irrelevant information about potentially selectable object during rasterization stage. Enforcing of such compatibility rule via M connector would ensure correct dynamic setup and integration of the modules.

As it can be concluded from the description, the functionality of the framework greatly depends on the content of available artifacts in repositories and correct definition of interdependencies between these. The proposed model is based on the assumption that the content will be gradually developed by third-party providers and filled in by the framework manager corresponding to the plug-in concept.

IV. Evaluation metrics

The general problem with evaluation of functional performance of any software system is that in most cases it is highly subjective and depends on personal preferences of each potential user. In general, formalization of such general metrics as "usability" would be possible only in case if it is divided into a set of low level metrics, each with precise definition, allowed value types, and ranges. Previous research of the authors of this paper in the area of evaluation of GVS can be found in [5].

Within the scope of the proposed framework, evaluation task becomes more structured and precise due to presence of RMM connectors that allow tracing each evaluation metric to corresponding requirement (or even multiple requirements, in case many-to-many relationships were introduced).

In most cases, development of a metric artifact must be done simultaneously with the development of requirement artifact as these are semantically related. For example, if the user introduces requirements "Support of three-dimensional graph drawing" and "Support of orthogonal layout" an appropriate metrics artifact would be "The maximal number of graph edges a single node may have" with allowed interval [0, 6]. In case in resulting implementation it is more than six, the condition will not be met and the overall evaluation of this aspect will become negative, leading to the necessity to return to requirements or assembly stages. As it was mentioned previously, each metric must be mapped to standard interval [0, 100].

Within our framework we propose to support the following 5 general types of metrics:

- A single Boolean value that defines whether particular requirement was met (in our case – yes/no). Default value: "yes". An example: "Ability to mark and distinguish graph elements with different color". During evaluation "Yes" corresponds to "100", "No" corresponds to "0".
- Structured hierarchical tree of Boolean values for complex requirement properties. Default value: all sub-nodes set to "yes". An example: "Ability to select multiple nodes and edges simultaneously" – the first level "Ability to select multiple nodes" with sub-nodes {"Two", "More"}; the second level "Ability to select multiple edges" with same sub-

nodes. During evaluation the average value between all levels is calculated.

- A single number value within allowed interval (minimum / maximum). Default value: maximum. An example – see above. During evaluation result must be interpolated and linearly scaled to [0, 100].
- Enumerated list of states that can be mapped to numbers. Default value: first from the list. An example: {"not implemented", "partially implemented", "almost completely implemented", "completely implemented"}. During evaluation the result must be interpolated and linearly scaled to [0, 100]. If provided values do not correspond to linear distribution, the second layer with appropriate weights must be supplied (for example [0, 0.33, 0.66, 1]) for correct interpolation.
- Enumerated list of abstract strings – all conditions same as above;

This list of types of metrics covers all primitive data types – Boolean, number (both integer and floating-point) and character / string. Moreover, each of these metrics can be presented either as an individual element or an array of the same type – this adds support of multidimensional collections (for example, an array of enumerated list of states).

The only common data type that is not included in this list is the reference ("pointer") to another value, but in this particular framework it has no use for evaluation of GVS, because the model of questionnaire requires from the participant to establish connections between sets of questions and answers, not between answers themselves.

In any case, a single united GVS evaluation rating can be easily calculated from the proposed types of metrics by averaging and if necessary – interpolating collected numerical values. This implies that the quality of the validation stage also depends on the content and quality of artifacts stored within framework repositories. This is another reason for maintaining a well-defined list of types – potential increase in the number of artifacts during the evolution of the framework requires more formal definition and management approach.

V. Related works

Software factoring in its modern interpretation is a relatively new IT domain that still experiences changes as new technologies and construction methods (for example – such as MDA) emerge. Although the concept itself appeared approximately in the 70-ies of the 20-th century, its implementation architecture is not stable.

According to the authors' knowledge, no other framework exists that is dedicated to construction of GVS, at least, none is implemented. Still, there are a number of solutions that provide general software factoring capabilities. Most of them are based on Microsoft .NET technology and incorporate such features as support of client/server architecture, Web ASP.NET application (n-tier architecture), SQL server / Oracle persistence, code generation for text based languages, reusable template engine, support for standard OOP patterns (collections, generics, inheritance), multilingual support, etc.

One such solution is provided as a part of „CodeFluent" project. Its common architecture for MS Windows applications is presented in Fig. 5, part (a) [8]. Another Microsoft interpretation of software factory architecture is presented in Fig. 5, part (b) [9]. Software factory for Web is presented in [10].

(a) (b)

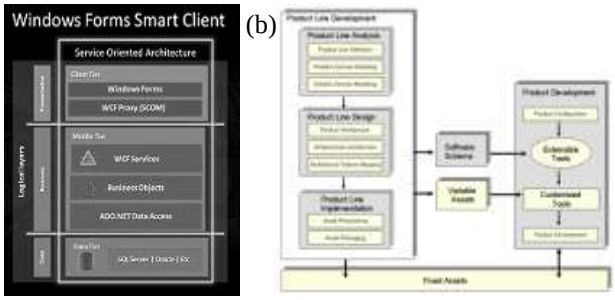

Fig. 5. Existing software factory architectures.

Although names may differ ("product line analysis", "product line design", etc.), in all cases there is an obligatory separation of the framework into levels, each dealing with an individual aspect of software life cycle. This is the general similarity of any software factory, regardless of its purpose.

The specifics of implementation, however, may vary. We believe that our framework with explicitly defined links between artifacts, dynamic configuration, and formal validation is sufficient for automated generation of GVS.

VI. Conclusions

In this paper the authors presented a description of the architecture of software factory that is capable of generation of GVS systems. The presented framework ensures full automation of gathering of requirements for the system being built and assembling of appropriate software modules. Validation is partially automated – it is carried out by the user, but the overall rating (conformance to initial requirements expressed in percents) is calculated based on the received input.

Demonstration of the proposed concept was based on capabilities of classic relation database and scripting language Lua in conjunction with general purpose language C++. This is not a mandatory requirement – actual implementation can be based on other solutions that ensure same capabilities.

The quality of GVS being generated depends on the available set of artifacts that is present in the repositories and quality of relationships defined between these. Although there are no particular requirements for the experience of the end user who performs specification and validation of GVS, there are additional requirements for the manager of the framework, who must fully comprehend its architecture.

Analysis of existing solutions in this domain revealed that no other semantically close frameworks for visualization had been implemented, although existing general purpose systems utilize similar approach to the automation of software lifecycle.

While continuing this research, it would be necessary to evaluate available technologies and choose particular solutions to integrate these in a prototype of the proposed framework

that would serve as a testing platform for further experiments with automated building of GVS.

REFERENCES

[1] B. Langlois, D. Exertier, "MDSoFa: a Model-Driven Software Factory," OOPSLA 2004, MDSD Workshop. October 25, 2004.
[2] P. Liegl, D. Mayrhofer, "A Domain Specific Language for UN/CEFACT's Core Components," services-2, pp.123-131, World Conference on Services - II, 2009.
[3] V. Zabiniako and P. Rusakov, "Development of Lightweight Software for Graph Drawing," GraVisMa 2010 workshop on Computer Graphics, Computer Vision and Mathematics, Brno, Czech Republic, 2010 – to be published.
[4] "Lua 5.1 Reference Manual" 2010. [Online]. Available: http://www.lua.org/manual/5.1. [Accessed: Oct. 1, 2010].
[5] V. Zabiniako and P. Rusakov. "Development and Implementation of Partial Hybrid Algorithm for Graphs Visualization," Scientific Proceedings of Riga Technical University, Computer Science, Applied Computer Systems, ser. 5, vol. 34, pp. 192 – 203, 2008.
[6] V. Zabiniako and P. Rusakov. "Supporting Visual Techniques for Graphs Data Analysis in Three-Dimensional Space," The 50th Scientific Conference of Riga Technical University, Computer Science, Applied Computer Systems, October, Riga, Latvia, 2009 – submitted for publication.
[7] V. Zabiniako and P. Rusakov. Definition of General Requirements for Graph Visualization Software," Scientific proceedings of Riga Technical University, Computer Science, Applied Computer Systems, ser. 5, vol. 38, pp. 168-179, 2009.
[8] "CodeFluent Entities – Features" 2010. [Online]. Available: http://www.codefluententities.com/Features.aspx. [Accessed: Oct. 1, 2010].
[9] J. Greenfield, K. Short, S. Cook, S. Kent and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, 2004.
[10] "Web Client Software Factory" 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb264518.aspx [Accessed: Sept. 27, 2010].

Vitaly Zabiniako was born in Riga, Latvia in 1983 and currently undertakes doctoral studies at Riga Technical University at the Faculty of Information Technology and Computer Science. Master's degree in Computer Science at Riga Technical University, 2007.

Systems analyst in the private IT company "ABC Software", previously – System Administrator at Riga Technical University, Institute of Applied Computer Systems. Selected previous publications: V. Zabiniako, P. Rusakov. Comparative Analysis of Visualization Aspects in Technologies Direct3D and OpenGL. Scientific Proceedings of Riga Technical University, Computer Science, series 5, vol. 26, pp 209-221 (2006). Interests and dissertation are connected with computer graphics and information visualization.

Awards – diplomas with distinction (Bachelor of engineering science in computer control and computer science; Master of engineering science in computer systems).

Pavel Rusakov was born in Riga, Latvia in 1972. Dr. sc. ing. (1998), Mg. sc. ing. (1995), Bc. sc. ing. (1993) – Riga Technical University (RTU).

Associated professor at RTU, Institute of Applied Computer Systems. Head of laboratory, responsible for the Professional Bachelor and Professional Master Studies at the Department of Applied Computer Science. Field of interest: computer science. Special interests: programming paradigms, object-oriented approach to systems development, parallel computing, Web technologies, distributed systems, computer graphics, and protection of information.

Diploma with distinction: Mg. sc. ing.

Vitālijs Zabiņako, Pāvels Rusakovs. Ietvara definēšana automātiskai grafu vizualizācijas sistēmu veidošanai.
Raksta ietvaros autori veica analīzi ar mērķi piedāvāt un novērtēt oriģinālu automātiskās trīsdimensiju grafu vizualizācijas sistēmas izveidošanas koncepciju, kā arī izstrādāt to soļu specifikāciju, kuri ir nepieciešami attiecīgā ietvara veiksmīgai konstruēšanai. Rakstā ir aprakstīts pilnais process – sākot ar prasību definēšanu grafu vizualizācijas sistēmai, tās automātisko konstruēšanu un beidzot ar iegūtā rezultāta novērtēšanu. Šie etapi ir reprezentēti triju līmeņu modeļa veidā, kur katrā no līmeņiem pastāv savstarpēji saistītie elementi, kuri nodrošina korektu procesa automatizāciju. Ir sniegts katra konstruēšanas soļa detalizētais apraksts, kopā ar papildu komentāriem par tiem aspektiem, kuri attiecas uz papildu informācijas iegūšanu no lietotāja (gadījumos, kad attiecīgi dati var būt izmantoti ģenerējamās sistēmas kvalitātes paaugstināšanai). Ar raksta autoriem ir izstrādāta un piedāvāta novērtēšanas metriku kopa, kura nodrošina atšķirības starp iecerētas un automātiskās konstruēšanas gaitā iegūtas vizualizācijas sistēmas funkcionalitātes formālo novērtēšanu. Metriku kopas saturs ļauj nosegt visus iespējamus datu tipus, kuri var būt noderīgi lietotājam grafu vizualizācijas sistēmas novērtēšanas gaitā. Lai būtu iespēja demonstrēt šāda veida arhitektūras realizācijas iespējas, tiek piedāvāts izmantot objektorientētās programmēšanas valodas „Lua" un „C++", kā arī relāciju datu bāzes pārvaldības sistēmu, specificējot visas nepieciešamas tabulas, attiecības starp tām, kā arī procedūras piekļuvei datiem. Ir aplūkoti pašreiz eksistējoši programmatūras konstruēšanas risinājumi „CodeFluent" un „Web Client Software Factory", kuri tiek salīdzināti ar piedāvātā ietvara arhitektūru. Minētas salīdzināšanas mērķis – identificēt potenciālās atšķirības, kā arī iezīmēt vispārīgus principus šāda tipa programmatūras sistēmu konstruēšanā. Raksta noslēguma daļā ir izklāstīti secinājumi, kuri bija saņemti darba ietvaros, kā arī ir definēti nākamie potenciālie pētījumi automātiskās grafu vizualizācijas sistēmu konstruēšanas jomā.

Виталий Забиняко, Павел Русаков. Определение архитектуры для автоматического создания систем визуализации графов.
В рамках статьи авторы выполнили анализ, цель которого – предложить и оценить оригинальную концепцию автоматического создания систем визуализации трехмерных графов, а также разработать спецификацию шагов, необходимых для успешного создания соответствующего каркаса. В статье описан полный процесс – начиная с определения функциональных требований к системе визуализации графов, описания её автоматического конструирования, и заканчивая оценкой полученного результата. Рассматриваемые этапы представлены в виде трехуровневой модели, где каждый из уровней содержит взаимосвязанные элементы для обеспечения корректной автоматизации процесса. Произведено детальное описание каждого шага процесса конструирования, даны комментарии по аспектам, связанным с получением от пользователя дополнительной информации (в случаях, когда соответствующие данные уместны для повышения качества автоматически генерируемой системы). Авторы предлагают набор типов метрик, обеспечивающий формальную оценку разницы между изначально задуманной и полученной в процессе автоматического конструирования системой визуализации. Данный набор метрик содержит все необходимые типы данных, которые могут быть полезны пользователю для оценки системы визуализации. Для демонстрации возможностей реализации данной архитектуры предложено использовать объектно-ориентированные языки программирования "Lua" и "C++", а также реляционную систему управлениями базами данных, определяя все необходимые таблицы, отношения между ними, а также процедуру доступа к данным. Рассмотрены существующие решения автоматического конструирования программного обеспечения „CodeFluent" и „Web Client Software Factory", которые сравниваются с предложенной архитектурой. Цель данного сравнения – идентифицировать потенциальные различия, а также выделить общие принципы автоматического создания систем программного обеспечения подобного рода. В заключительной части статьи изложены выводы, а также определены дальнейшие возможные исследования в области создания систем визуализации графов.