Domain Specific Languages for Business Process Management: a Case Study

Janis Barzdins Karlis Cerans Mikus Grasmanis
Audris Kalnins Sergejs Kozlovics Lelde Lace Renars Liepins
Edgars Rencis Arturs Sprogis Andris Zarins
Institute of Mathematics and Computer Science, University of Latvia
Raina blvd. 29, LV-1459, Riga, Latvia
{janis.barzdins, karlis.cerans, mikus.grasmanis, audris.kalnins, sergejs.kozlovics, lelde.lace, renars.liepins, edgars.rencis, arturs.sprogis, andris.zarins}@lumii.lv

ABSTRACT
Nowadays, more and more issues need to be considered when implementing tools for domain specific languages with an orientation to the business process management. It is not enough to build just an editor for the language, various specific services need to be provided as well. In this paper, we describe our approach how to develop new domain specific languages for the mentioned field and their support tools. A description of, so called, transformation-driven architecture is outlined as well. It is shown how to use principles of the architecture in developing tool building platforms. Two domain specific languages together with tools implementing them are described as an example.

Keywords
Transformation-Driven Architecture, tool building platform, metamodels, model transformations, business process management.

1. INTRODUCTION
When talking about business processes and their role in development of information systems, an abbreviation BPM usually comes up. However, the meaning of BPM is not always the same. Initially, the letter M stood for Modeling, so with BPM everyone was to understand the development of tools being able to design business processes graphically. Later, the modelers' community realized it is not enough, and the second meaning of BPM arose – Business Process Management [1]. Now, business processes are not only modeled but also managed (meaning the process modeling tool had been integrated into some process management system which controls the process execution and integrates other parts of the information system).

Consequently, two kinds of graphical languages regarding business processes exist nowadays. Firstly, there are plenty of business process modeling languages. One of the most popular of them is probably the UML [2] activity diagrams. And secondly, there are also some business process management languages for which a compiler to some code executable on a process engine exists. Here, one must mention the BPMN (Business Process Modeling Notation [3]) and its possible target language – the BPEL (Business Process Execution Language [4]). A very important component of a BPMN tool is a compiler to the BPEL code being executable by some BPEL engine. According to the SOA ideology [1], a web service wrapper is developed for the information system components allowing the BPEL engine to manage execution of the whole system being allowed to be distributed through multiple companies.

However, most of these BPM languages or tools are often not very useful in everyday situations. Being very complex they are of course very useful for large enterprises. However, smaller and more specialized systems usually need only a small part of those facilities provided by the universal languages and tools. As a result, the usage of them tends to be too complicated. Moreover, tools (called the BPM suites) providing efficient and reliable implementation of process management languages and offering a whole set of support facilities are basically very expensive. Certainly, there are also some less expensive suites (e.g., BizAgi [5]) and cheap software-as-service offers by other vendors (e.g., Intalio [6]), but they are based on the same complicated languages and approaches. Therefore, specialized languages for narrow business domains are required, and that is where the DSLs (Domain Specific Languages) come into play. Although universal languages make advances towards specific tool builders (e.g., BPMN offers a possibility to add new attributes for tasks), they can never give such wide spectrum of facilities as DSLs can. In addition, frequently there are already well accepted notations for manual design of processes in some business domains, and they can be adequately formalized by the DSL approach. Moreover, buying and adapting some universal language or tool for one's small and specific case can often overcharge the benefits of using it afterwards. On the other hand, the development of a DSL can give little benefit if its implementations cost much. So we do have a need for some simple and unified way of building domain specific languages and tools. In this paper, we present a method of developing and implementing domain specific languages. Also, a success story of implementing two concrete DSLs is described here.

The paper is organized as follows. In Section 2, some possible requirements for tools implementing domain specific languages have been discussed. Two example tools ordered by real customers are introduced as well. Since they are to be parts of some information systems, some concrete services were to be satisfied by the tool. In Sections 3 and 4, our solution is presented. Besides that, the most important aspects of our metacase tool's

architecture are sketched here as well. The approach of the architecture is demonstrated on the mentioned example tools. Section 5 concludes the paper.

2. TYPICAL REQUIREMENTS FOR DSL TOOLS IN THE FIELD OF BPM

2.1 DSL tools in general

When developing a tool for a domain specific language (a DSL tool) ordered by a customer, various needs have to be satisfied usually. Generally speaking, a DSL tool consists of two parts – a domain specific language it implements, and services it offers. So, various tools differ one from another not only in the notation of a DSL, but also in extent how easy they can be integrated in the outer world. Nowadays, life does not end with an editor of some domain specific language, it just starts there. In general, various types of functionality must be provided when designing a domain specific language including (but not limited to) a compiler to some target environment, a simulation feature, a debugger etc. However, when designing a DSL in the field of business process management, some more specific features come to mind. For a BPM domain specific tool to be successful, it must be able, for instance, to establish a connection to some external data source, for instance, a relational database. A DSL editor is often supposed to be a part of some larger information system, so it must provide facilities of collaboration with other parts of the system, the database being one of them. Besides, the collaboration must be possible in both design and run time of the tool. A crucial feature is also the ability to convert a process definition in this DSL into specification for some process execution engine in the system.

Other important issue to be considered is the ability to generate some kind of reports from the model information. DSL editors are often used to ease the preparation of, e.g., HTML or Microsoft Word documents containing information about the domain. So the tool should provide a way of generating such documents from the user-drawn diagrams.

Considering these issues is a crucial factor when designing a new DSL tool building platform. Some of the key facilities can be designed easily and added to the platform. However, others can be added later when such a necessity occurs. So, trying to satisfy the needs of different customers can play a great role in the growth of the platform. Therefore, in the next sections, we offer a description of two domain specific tools and explain their implementation within our DSL tool building platform.

2.2 Example tools

In this section, two concrete domain specific languages together with the tools implementing them will be discussed. First of them – Project Assessment Diagrams (further – PAD) – is an editor for visualizing business processes regarding review and assessment of submitted projects. This editor is based on UML activity diagrams and thus contains means for modeling business processes. Yet, some new attributes and some new elements have been added in order to handle the specific needs. For example, elements for controlling execution duration have been designed (elements *SetTimer* and *CheckTimer* that can be attached to a flow). The PAD editor has to be a part of a bigger information system for document flow management (a simplified BPM suite), so services providing interconnection between the system and the editor were needed. For example, a PAD model needs to be imported in a database where the information system can, for instance, make a trace for each client's project and then project this trace back to the editor for the visualization. This requirement was in some way similar to the business process monitoring performed, e.g., in ARIS [7] where groups of reasonably selected instances can be monitored. They go even further – a process mining is introduced to automate the monitoring process. So again – the problem has been known for some time already, but here we are trying to solve it by the means of a DSL instead of a universal language. Also, we do not need such powerful features providing the whole mining process. Instead, a very simple solution for business process monitoring was requested here.

The other domain specific language (and tool) we have developed is an editor for business processes in the State Social Insurance Agency (further – SSIA). Since users' habits were to be taken into account, this language syntactically is closer to BPMN. Again, specific services needed to be satisfied by the tool, three of which are the most worth mentioning:

- Online collaboration with a relational database – the searching for information in a database was to be combined with the graphical tool. The use case of that was a possibility to browse for normative acts during the diagram design phase – the normative acts are stored in a database and need to be accessed from the tool.
- Users wanted to start using the tool as soon as possible – even before the language definition has been fully completed. That means we need to assure the preservation of user-made models while the language can still change slightly. So the DSL evolution over the time is an issue to be considered.
- The tool must be able to generate some kind of reports from the visual information, preferably – in the format of Microsoft Word. Moreover, some text formatting possibilities must be provided in the tool, e.g., by ensuring the rich text support to input fields.

Besides those, some more minor issues were highlighted during the DSL design phase, but we are not going to cover all of them here due to the space limitation.

It must be mentioned that, when designing languages, the main emphasis was put on the fact that processes must be easy perceived by the user. At the same time, however, languages had to be suitable for serving as process management languages without any changes. Since languages have been designed in such a manner, it is possible to integrate them into a full-scale BPM suite later. There the process definitions will be used to manage the document flows in a typical to BPM manner.

3. IMPLEMENTATION BACKGROUND

3.1 General ideas

We have used our metamodel-based Graphical Tool-building Platform GrTP [8] to implement the domain specific languages PAD and SSIA. The recent version of GrTP is based on principles of the Transformation-Driven Architecture (TDA, [9]). In this Section, the key principles of the TDA and GrTP as well as their applications in DSL implementation are discussed.

3.2 The Transformation-Driven Architecture

The Transformation-Driven Architecture is a metamodel-based approach for system (in particular, tool) building, where the system metamodel consists of one or more interface metamodels served by the corresponding engines (called, the interface engines) and the (optional) Domain Metamodel. There is also the Core Metamodel (fixed) with the corresponding Head Engine. Model transformations are used for linking instances of the mentioned metamodels (see Fig. 1).

The Head Engine is a special engine, whose role is to provide services for transformations as well as for interface engines. For instance, when a user event (such as a mouse click) occurs in some interface engine, the Head Engine may be asked to call the corresponding transformation for handling this event. Also, a transformation may give commands to interface engines. Thus, the Core Metamodel contains classes Event and Command, and the Head Engine is used as an event/command manager.

Since it has been published in [9], we won't go into details about TDA here. Instead, we will just outline the main technical assumptions for TDA in order to set the background:

- The model data are stored in some repository (like EMF [10], JGraLab [11] or Sesame [12]) with fixed API (Application Programming Interface).
- The API of the repository should be available for one or more high-level programming languages (such as C++ or Java), in which interface engines will be written.
- Model transformations may be written in any language (for instance, any textual language from the Lx family [13] or the graphical language MOLA [14] may be used). However, the transformation compiler/interpreter should use the same repository API as the engines.
- When a transformation is called, its behavior depends only on the data stored in the repository.
- Only one module (transformation or engine) is allowed to access the repository at the same time. Concurrency and locking issues are not considered.

We have developed a, so called, TDA framework which implements the principles of the TDA. The TDA framework contains one predefined engine – the head engine – and the repository (we are using our very efficient in-memory repository [15] with a fixed API being available from the programming language C++ in which engines are to be written). Other interface engines may also be written and plugged-in, when needed. The TDA framework is common to all the tool building platforms built upon the TDA. The framework is brought to life by means of model transformations. One can choose between writing different transformations for different tools and writing one configurable transformation covering several tools.

Actually, one more layer is introduced between the model transformations and the repository. It is called the repository proxy and it contains several features being common for all tool building platforms built upon the TDA. The most notable of them is perhaps the UNDO/REDO functionality – since it is embedded in the proxy, engines and transformations do not have to consider the UNDO and REDO actions. All the commands are intercepted by the proxy and then passed further to the repository.

3.3 The TDA-based Tool Building Platform GrTP

Next, we have developed a concrete tool building platform called the GrTP by taking the TDA framework and filling it with several interfaces. Besides the core interface, five more interfaces have been developed and plugged into the platform in the case of GrTP:

- The graph diagram interface is perhaps the main interface from the end user's point of view. It allows user to view models visually in a form of graph diagrams. The graph diagram engine [16] embodies advanced graph drawing and layouting algorithms ([17, 18]) as well as effective internal diagram representation structures allowing one to handle the

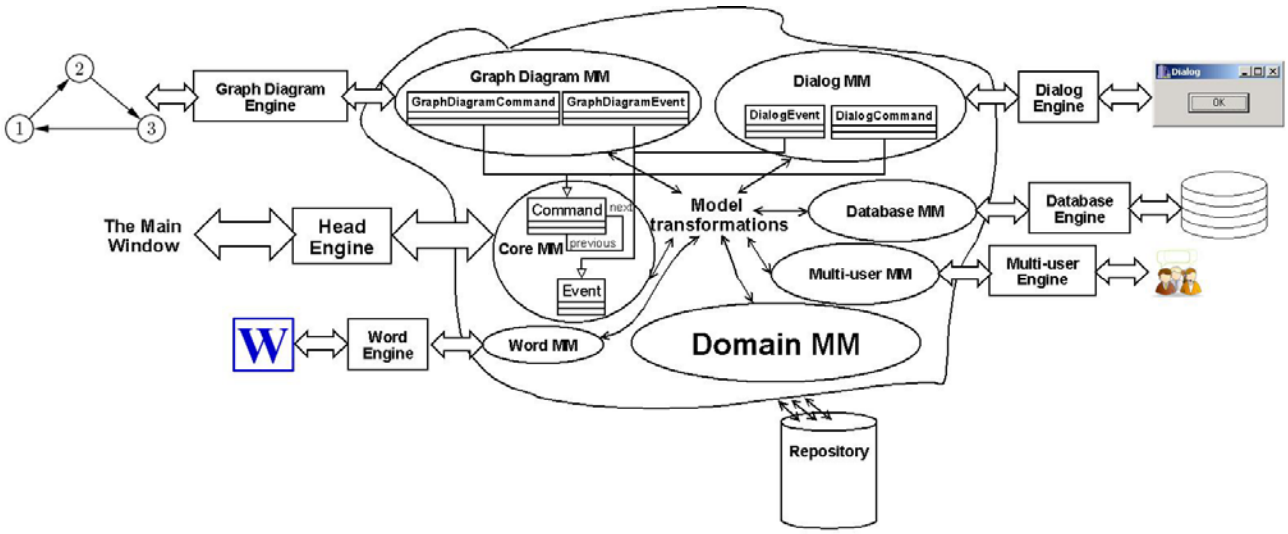

Figure 1. The Transformation-Driven Architecture framework filled with some interfaces.

visualization tasks efficiently even for large diagrams.

- The property dialog interface allows user to communicate with the repository using visual dialog windows.
- The database interface ensures a communication between the model repository and a database.
- The multi-user interface performs the task of turning a project into a multi-user project and considers other issues regarding that.
- The Word interface helps user to establish a connection to Microsoft Word and to send data to it.

The final step is to develop a concrete tool within the GrTP. This is being done by providing model transformations responding to user-created events. A fair part of these transformations usually tend to be universal enough to be taken from our already existing transformation library instead of writing them from scratch (transformations responding to main events like creating new element, reconnecting line ends, making a mouse click or double click etc., as well as such platform specific transformations as copy, cut, paste, delete, import, export, etc.). In order to reduce the work of writing transformations needed for some concrete tool, we introduce a tool definition metamodel (TDMM) with a corresponding extension mechanism. We use a universal transformation to interpret the TDMM and its extension thus obtaining concrete tools working in such an interpreting mode. This is explained a bit more in the next subsection.

3.4 The tool definition metamodel and its usage for building concrete tools

First of all, we explain the way of coding models in domain specific languages. The main idea is depicted in Fig. 2. As can be seen here, the graph diagram metamodel (conforming the one from Fig. 1) is complemented with types turning a general graph diagram into a diagram of some concrete tool (e.g., some business process editor). A model here is a set of graph diagrams every one of which consists of elements – nodes and edges. An element in its turn can contain several compartments. At runtime, each visual element (diagrams, nodes, edges, compartments) is attached to exactly one type instance (see classes *DiagramType*, *ElementType*, *CompartmentType*) and to exactly one style instance. Here, types can be perceived as an abstract syntax of the model while the concrete syntax being coded through styles.

Now, about the proposed tool definition metamodel. The main idea of the tool definition metamodel together with the extension mechanism is presented in Fig. 3. Apart from types, the tool definition metamodel contains several extra classes describing the tool context (e.g., classes like *Palette*, *PopUp*, *ToolBar*, etc.). Moreover, the tool definition metamodel contains, so called, extension mechanism providing a possibility to change behavior of tools represented by the metamodel. The extension mechanism is a set of precisely defined extension points through which one can specify transformations to be called in various cases. One example of a possible extension could be an "AfterElementCreated" extension providing the transformation to be called when some new element has been created in a graph diagram. Tools are being represented by instances of the TDMM by interpreting them at runtime. Therefore, to build a concrete tool actually means to generate an appropriate instance of the TDMM and to write model transformations for extension points. In such a way, the standard part of any tool is included in the tool definition metamodel meaning that no transformation needs to be written for that part. Instead, an instance of the TDMM needs to be generated using a graphical configurator. At the same time, the connection with the outer world (e.g., a database or a text processor) is established by writing specific model transformations and using the extension mechanism to integrate them into the TDMM.

3.5 Benefits of the TDA

The main advantage of the transformation-driven architecture is its idea of providing explicit metamodeling foundations in building tools for domain specific languages. Although there

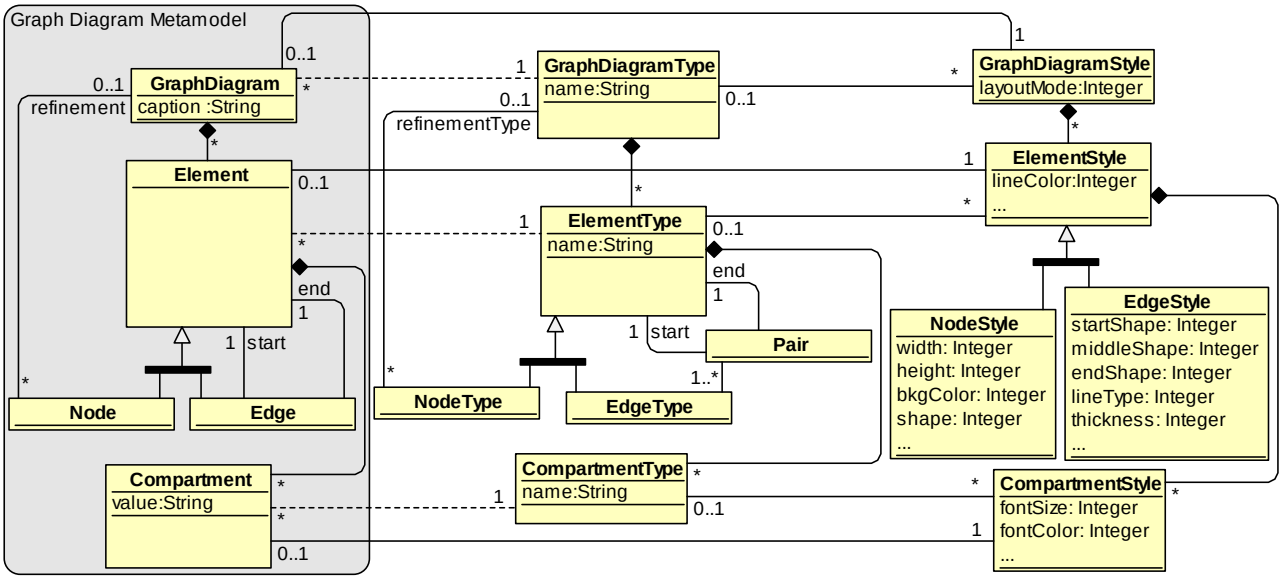

Figure 2. The way of coding models.

already exist some metacase tools accepting the idea of DSL tool definition by a metamodel (e.g., Eclipse GMF [19] and Microsoft DSL Tools [20]), they generally offer only some configuration facilities allowing definition of a DSL tool in a user-friendly way while a direct access to metamodels is either limited or provided using some low level facilities. The pace-maker of the field is perhaps the Metacase company whose product MetaEdit+ [21] provides Graph, Object, Property, Port, Relationship and Role tools to ensure the easy configuration of concrete domain specific tools. Another well-known example is Pounamu/Marama [22, 23] which offers shape designer, metamodel designer, event handler designer and view designer to obtain a DSL tool. On the contrary, the TDA is completely transparent meaning a user can have a free read and write access to its metamodels and their instances. Of course, extra services like graphical configurator of a DSL tool can be offered as well, but the user is not forced to use it. It must be underlined that, if following the TDA, the definition of a concrete domain specific tool only involves developing model transformations and nothing else. The TDA follows the ideas of the MDA [24] stating that the common part of syntax and semantics can be formalized through a metamodel. The whole specific part at the same time can be put into model transformations.

The other notable advantage of the TDA is its ability to get in touch with the outer world. This is being done by adding new engines to the TDA framework. Since there is no need to go deep in implementation details of other engines or other parts of the TDA, this is considered to be a comparatively easy task.

4. The development of PAD and SSIA using GrTP

Besides the trivial part – generation of a tool definition metamodel's instance forming the graphical core of the tool – we decided to develop three more engines we did not have at that moment. Those engines were the database engine, the multi-user engine and the Word engine. Since the TDA framework provides a possibility to plug in new interfaces (engines together with their respective metamodels) easily, the development and integration of the engines was done quite harmlessly. We must admit there were some difficulties to integrate the multi-user interface, however they were mostly of technical nature – the tool definition metamodel had to be changed a bit as well.

Next, according to the extension mechanism, some specific transformations needed to be written in order to put a life into the static tools – to make them dynamic. These transformations referred to generating, for instance, the correct items for combo boxes, to changing items in context menus dynamically, to assigning the correct styles to visual elements (although this can be partly specified in the static part as well) etc. These transformations had to be written and attached to appropriate

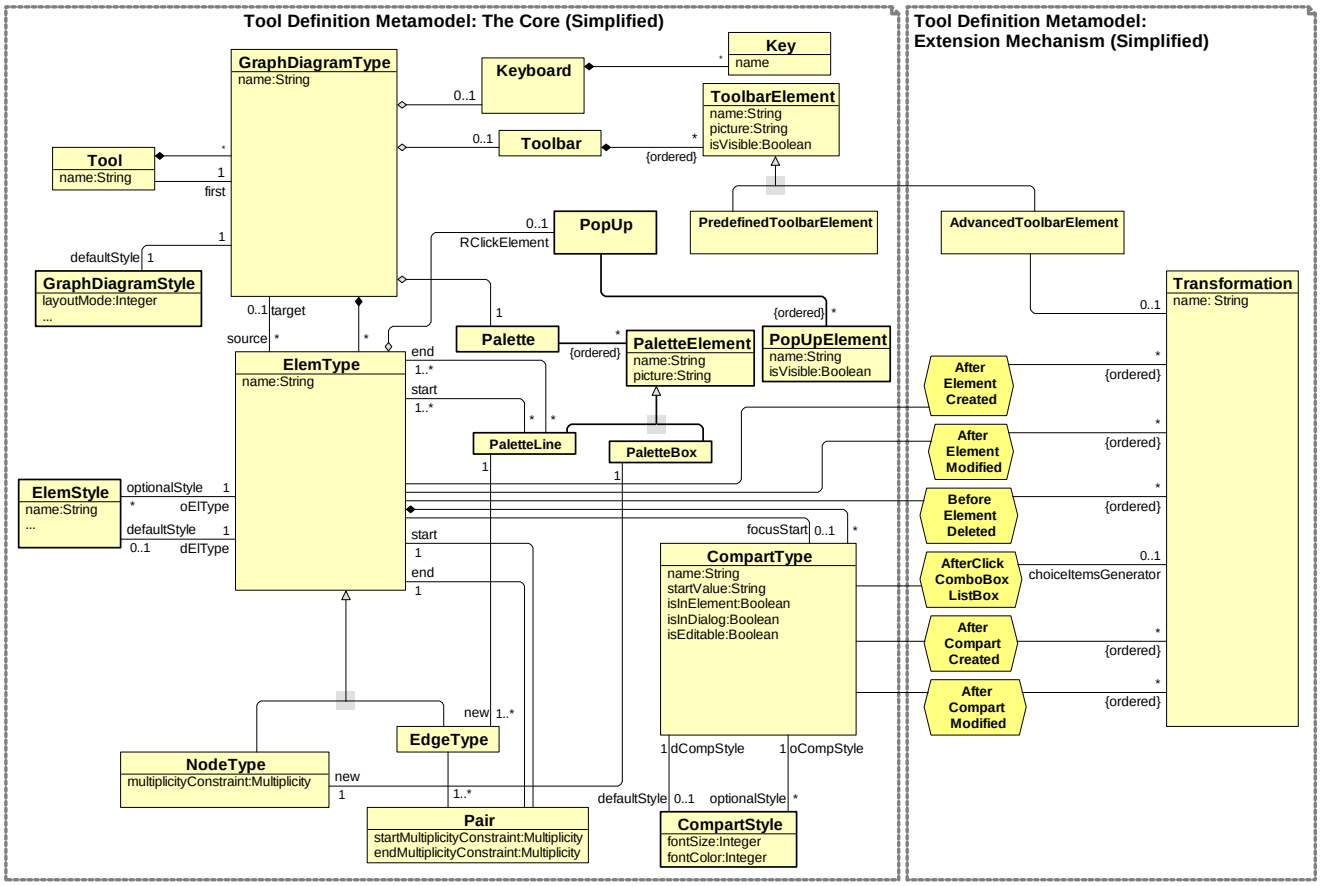

Figure 3. Basic principles of the tool definition metamodel.

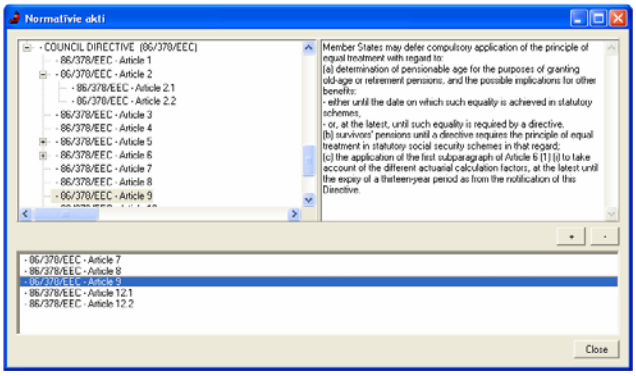

Figure 4. Browsing for normative acts stored in the database from the tool's dialog window.

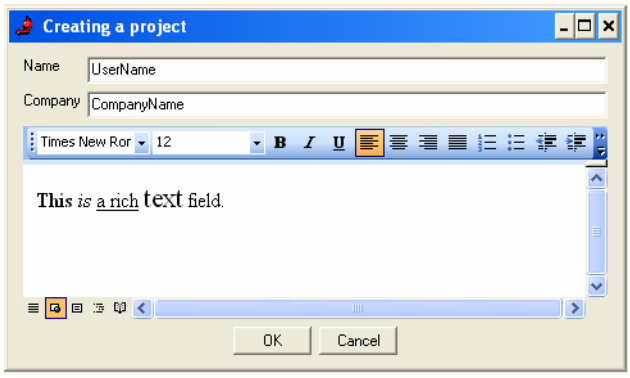

Figure 5. A rich text component.

extension points thus forming the concrete tool.

The most challenging part of the development of tools was perhaps the ensuring interconnection between the tool and a relational database. Since the graphical tool was meant to be just one piece of the whole information system's software, it was already clear before that this problem will have to be faced sooner or later. The issue has been classic in the world of workflows – some business process has been being modeled in a tool and passed to a relational database afterwards. The information system would then take care of applying the process to individual clients and storing the history of how far each client has gone through the process. On request, the tool should be able to visualize the history for some particular client as well.

The classic solution of the problem advises using the ORM method (Object-relational mapping, [25]) stating that a simple mapping between the model repository and the database must be made and a generation of metamodel instances and/or database records must be performed. However, this solution was not acceptable in our case because of its limitations in the process of generation of instances as a response to a query for the database – the types or return tables must be known before. At the same time, the SSIA project required the possibility of receiving answer to an arbitrary query.

Our solution included turning a part of the metamodel storing the business processes together with their respective element types and styles (see Fig. 2) into a database schema. That was a pretty straightforward job – if abstracting from the details, the database was made to store data in RDF format [26]. So, all the database engine had to do was generating the contents of the database from the model and vice versa. Next, a translator was made in an information system part of the system carrying out a connection between the RDF-type database and the actual database of the system. Thus, by introducing such an intermediate layer between the tool and the actual database, the database engine was to be written once and for all – it does not depend on the actual database schema.

Eventually, the tools obtained in GrTP satisfied all the needs customers had highlighted, including the ones mentioned in Section 2. The connection to the relational database provided by the database engine ensured fast information searching capabilities in database in combination with the tool. Thus, an easy browsing for information stored in the relational database (in this case – normative acts) was possible from the tool interface (see Fig. 4). Next, an add-only DSL evolution comes at no extra cost if using the tool definition metamodel to develop tools in GrTP. Indeed, if the DSL demands some more element or compartment types to be added, we just add new instances to *ElementType* or *CompartmentType* classes in the model coding metamodel. Since it has nothing to do with already existing types of the language, existing models remain unmodified. This can be achieved because of the fact that we store the DSL definition in the same modeling level with the actual models – the connection between a model element and its definition is obtained without crossing levels. However, if changes in DSL are not of add-only type, some extra work needs to be done – elements and compartments of old types may need to be either deleted of relinked to some new types (see dashed associations in Fig. 2). In our framework, all this work can be done by model transformations. It must be mentioned that add-only changes are comparatively easy to implement in most metacase tools, although not in all. For example, it is still a quite tough problem in tools based on JGraLab repository [11].

Finally, a report generator was built using the Word engine. It introduces a simple graphical language allowing one to specify the information to be put in a Microsoft Word document. In the engine, several extra services were implemented as well. For example, a Word window was embedded in a property dialog windows generated by the property dialog engine and providing a possibility for a user to create rich text compartment values as was requested by the customer (see Fig. 5).

5. CONCLUSIONS

In this paper, we described our approach how to develop new domain specific languages and tools for supporting them. A short description of the transformation-driven architecture and its framework was outlined as well. The architecture was illustrated in its application in the graphical tool building platform GrTP upon which two concrete domain specific languages were implemented.

It was mentioned that nowadays DSL is often to be only a part of some bigger information system and the tool supporting it thus must be able to communicate with the outer world. In fact, this approach is only one of the possible solutions for the problem of how to integrate the tool with other parts of an information system. The other possible way is to develop so called business process management suites in which all the necessary features are

included. It involves also the issue of how to include fragments of existing information systems into a tool environment. It is usually done by turning components of the information system into a web service thus providing an appropriate network addressable application program interface for it.

Considering the issues mentioned above, our closest goal is to develop a platform for building domain specific suites. One of the possible domains for the approach could be suites incorporating process and document management integrated with sophisticated document generation procedures. Though large-scale expensive solutions such as EMC Documentum exist here, a very appropriate niche for small-scale, but logically sophisticated DSL-based solutions could be document management in various government institutions. The web service based approach will ensure very tight integration of the DSL execution environment with the rest of the suite including full access to databases.

6. REFERENCES

[1] J. F. Chang. Business Process Management Systems. Auerbach Publications, 2006, pp. 286.

[2] UML, http://www.uml.org.

[3] BPMN, http://www.bpmn.org.

[4] BPEL, http://www.bpelsource.com.

[5] BizAgi, http://www.bizagi.com.

[6] Intalio, http://www.intalio.com.

[7] ARIS platform, http://www.ids-scheer.com/en/ARIS_ARIS_Software/3730.html.

[8] J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation Based Graphical Tool Building Platform. *Proc. of Workshop on Model Driven Development of Advanced User Interfaces, MODELS 2007*, Nashville, USA.

[9] J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, USA, 2008, pp. 60–63.

[10] Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf.

[11] S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek f¨ur TGraphen, *Diplomarbeit*, University of Koblenz-Landau, Institute for Software Technology, 2006.

[12] Sesame, http://www.openrdf.org, 2007.

[13] J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and their Implementation by Bootstrapping Method. *Pillars of Computer Science*, LNCS, vol. 4800, Springer-Verlag, 2008, pp. 130-145.

[14] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004*, LNCS, vol. 3599, Springer-Verlag, 2005, pp. 62-76.

[15] J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks. Towards Semantic Latvia. *Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006)*, Vilnius, 2006, pp. 203-218.

[16] J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009*, Sanibel Island, Florida, USA, 2009, pp. 29-32.

[17] P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proceedings of Graph Drawing '95*, LNCS, vol. 1027, Springer-Verlag, 1996, pp. 361–364.

[18] K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proceedings of The Latvian Academy of Sciences*, Section B, vol. 55, No. 1, 2001, pp. 43–51.

[19] A. Shatalin, A. Tikhomirov. Graphical Modeling Framework Architecture Overview. *Eclipse Modeling* Symposium, 2006.

[20] S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools, Addison-Wesley, 2007.

[21] MetaEdit+, http://www.metacase.com.

[22] N. Zhu1, J. Grundy, J. Hosking. Pounamu: a meta-tool for multiview visual language environment construction. *Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04)*, 2004, pp. 254-256.

[23] J. Grundy, J. Hosking, N. Zhu1, N. Liu. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. *21st IEEE International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 25-36.

[24] MDA Guide Version 1.0.1. OMG, http://www.omg.org/docs/omg/03-06-01.pdf.

[25] C. Richardson. POJOs In Action. Manning Publications Co, 2006, pp. 560.

[26] Resource Definition Framework, http://www.w3.org/RDF.