9. Sutcliffe, A.: On the inevitable intertwining of requirements and architecture. Lecture Notes in Business Information Processing **14** (2009) 168–185
10. Barber, K.S., Graser, T.J., Grisham, P.S., Jernigan, S.R.: Requirements evolution and reuse using the systems engineering process activities (sepa). Australian Journal of Information Systems **7**(1) (1999) 75–97 Special Issue on Requirements Engineering.
11. Zowghi, D., Gervasi, V.: On the interplay between consistency, completeness, and correctness in requirements evolution. Information and Software Technology **45** (2003) 993–1009
12. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Transactions on Software Engineering **31**(7) (2005) 551–569
13. Śmiałek, M.: From user stories to code in one day? Lecture Notes in Computer Science **3556** (2005) 38–47 XP 2005.
14. de Boer, R.C., van Vliet, H.: On the similarity between requirements and architecture. Journal of Systems and Software **82**(3) (2009) 544 – 550
15. Du Bois, B., Demeyer, S.: Accommodating changing requirements with EJB. Lecture Notes in Computer Science **2817** (2003) 152–163
16. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading (1992)
17. Object Management Group: Unified Modeling Language: Superstructure, version 2.2, formal/2009-02-02. (2009)
18. van den Berg, K.G., Simons, A.J.H.: Control flow semantics of use cases in UML. Information and Software Technology **41**(10) (1999) 651–659
19. Śmiałek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Complementary use case scenario representations based on domain vocabularies. Lecture Notes in Computer Science **4735** (2007) 544–558
20. Kaindl, H., Śmiałek, M., Svetinovic, D., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Schwarz, H., Bildhauer, D., Brogan, J.P., Mukasa, K.S., Wolter, K., Krebs, T.: Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project (2007) www.redseeds.eu.
21. Śmiałek, M., Bojarski, J., Nowakowski, W., Straszak, T.: Scenario construction tool based on extended UML metamodel. Lecture Notes in Computer Science **3713** (2005) 414–429
22. Śmiałek, M.: Accommodating informality with necessary precision in use case scenarios. Journal of Object Technology **4**(6) (2005) 59–67
23. Mukasa, K.S., Jedlitschka, A., Graf, C., Klöckner, K., Eisenbarth, M., Steinbach-Nordmann, S.: Requirements specification language validation report. Project Deliverable D2.5.1, ReDSeeDS Project (2007)

Domain Specific Languages and Tools

Domain Specific Business Process Modeling in Practice

Janis Bicevskis, Jana Cerina-Berzina, Girts Karnitis, Lelde Lace, Inga Medvedis, Sergejs Nesterovs

University of Latvia, Raina blvd. 19, Riga, Latvia
Girts.Karnitis@lu.lv

Abstract. Experience of practitioners in modeling with Domain Specific Languages (DSLs) is analyzed in this paper. It is shown, that unlike general-purpose modeling languages (UML [1], BPMN [2]), DSLs provide means for concise representing semantics of the particular business domain that enable development of consistent and expressive business process models. Resulting models can be used not only as specifications for development of information systems, but also for generation of executable applications. Thus one of the principal goals of Model Driven Architecture (MDA [3]), the development of model-bases information system, is achieved.

Keywords: Business Process Modeling, BPM, Domain Specific Languages, DSL, Model Driven Architecture, MDA.

1 Introduction

A perfect information system, which is consistent with all customer requirements, reliable, easily adaptable and maintainable, still remains only a dream for IT developers. Main obstacle in the way to this dream is inability of customers, who mostly are not knowledgeable in information technologies, to define their requirements clearly and unambiguously, and to communicate them to the developers. Traditionally, information systems have been developed in compliance with some standardized documentation, for example, software requirements specifications, but, in practice, requirements, if they are formulated in natural language, tend to be inaccurate and ambiguous. The situation is even worsened by often changing customer requirements. All this places software developers in unenviable situation – they must develop software according to inaccurate and changing specifications.

A possible solution to the problem is to use a formal language to define requirements and to make a model for the system. This can eliminate ambiguity of requirements, and can even enable direct translation of specification into application. Thus, the problem of changeability of requirements becomes resolvable – changes can be introduced into model, and then application can be generated from the model.

However, despite decades-long efforts, these problems still are not completely solved. Traditional CASE technologies have given only partial results (see, for example, Oracle Designer [4]). Fierce competition in IT market demands information systems of exceptional quality, and available support from traditional CASE

technologies is not sufficient to provide adequate user interface, high usability, maintainability, performance and other quality requirements. Flexibility against changing requirements is still limited. For example, if requirements are changed, and these changes cannot be represented in the formal specification (because specification language is not sufficient), they must be incorporated directly into the source-code of the system, and, in case of automatically generated source-code serious problems can arise. CASE tools are also quite conservative and slow to catch up the fast development of the programming technologies producing new programming techniques and frameworks every year. As a result, only a fraction of applications can be generated from specifications.

IT experts are still looking for new ideas. One of the recent developments is Model Driven Architecture (MDA [3,5]). In order to make application development more flexible, this approach splits the process into two steps. First, the platform-independent model (PIM) is made in some general-purpose or domain-specific modeling language, for example, UML. Second, PIM is translated to a platform-specific model PSM, thus obtaining an executable application. This separation allows for more flexible generation of applications and development of user-friendly information systems. MDA approach evolves very fast, but some challenges are also at a glance.

General-purpose modeling languages, including UML, often used to make PIMs, are difficult to grasp for non-IT professionals, the future users of information systems. Even if they read and accept the models, their understanding is not deep enough, and they undervalue consequences of decisions behind these models. Code, generated from the PSMs, still does not produce usable and reliable software. Information systems are not flexible, and it is hard to achieve compliance with the models. If changes are made directly in the generated code, consequent re-generation can void them.

Trying to follow the path of MDA often ends with UML specifications, that is just one of the sources of information for developers of the software. Only in some projects UML models are directly translated into software ([6]).

Facing these problems in the practice again and again, and having advanced tool-building platform ([7]) at hand, we tried to solve them building domain-specific business process modeling tools, according to the following principles:

1. Tools must be comprehendible to non-IT specialists, because modeling will be done mostly by domain professionals.
2. Specific requirements of business domain and of information system development must be taken into account.
3. Modeling of real-life situations in the business domain must be possible, as well as ensuring and showing to users that information systems treat these situations correctly.

We have made a graphical domain-specific language, which, we hope, can easily be understood by non-IT professionals. This language is domain-specific: first, it can be used only for modeling of business processes, and, second, it can be used only in specific organizations. The language, called ProMod, is extended subset of on BPMN (see Chapter 2). Language deals mostly with behavioral aspects of business processes and applications and do not try to cover entire enterprise architecture as for example ArchiMate ([8]) do. Key feature of ProMod is that semantics of graphical primitives

is deeply specific for organizations where it is intended to be used. Business process model is a set of diagrams, interconnected with tree-like structures of enterprise data. Unlike many general-purpose modeling tools ensuring only syntactical correctness of models, ProMod provides tools for checking semantic consistency and completeness also. Consistency and completeness is checked with domain-specific rules – this kind of checking would not be possible in a general-purpose language.

Narrow usage domain of ProMod raises a question of amortization of efforts, i.e. whether the benefits gained are worth the time and money spent developing the language. Using transformation driven architecture to build DSL tools can solve this problem. We used metamodel-based graphical tool-building platform GrTP and it enabled us to create both graphical editor and consistency checker in a reasonably short time. This paper deals mostly with ProMod DSL – for detailed discussion of transformation-driven architecture and tool-building platform GrTP see [7], [9] and [10].

The paper is organized as follows. Chapter 2 contains description of our business domain and main features of ProMod DSL. Chapter 3 is a brief introduction to tool-building platform GrTP. Chapter 4 discusses current usage and ideas about future development of ProMod DSL.

2 Features of the Domain-specific Modeling Language

General-purpose modeling languages offer a fixed set of primitives: objects, attributes, connectors etc., with predefined semantics, that cannot be extended, or can be extended in some limited predefined manner (as, for example, stereotypes and profiles in UML [11], [12] or styling of graphics and option to adding attributes in BPMN [2], [13]). This kind of fixed notation is suitable for modeling in general, still much of the domain-specific information is represented in unnatural way or even remains outside the models. In the domain-specific languages we are looking for ways to extend the set of modeling primitives with ones, specific to the particular business domain ([14]).

2.1 Modelling Domain: State Social Insurance Agency

The domain, where we are looking for specific concepts, repetitive patterns and clichés of business organization to enrich the modelling language, is Latvian State Social Insurance Agency (SSIA) – a government institution, providing social insurance services: pensions, benefits, allowances etc. Like in many government institutions, all activities in SSIA are strictly prescribed by legislation and local instructions, but, unlike most government institutions, SSIA is a client-oriented enterprise – its main function is to service clients, and it has well-developed front-office. Servicing clients in a vast majority of cases means processing documents: for example, a client claims for some kind of social service and provides appropriate documents, these documents go thru a workflow, and in the end approval or rejection letter is sent to the client.

The domain of social security is sensitive sphere in Latvia and it is a target of frequent political decisions and new regulations resulting in frequent changes of information systems.

SSIA has recognized need for the management of business processes. Many of the business processes have already been defined in richly annotated IDEF0 ([15]) diagrams. Currently SSIA is planning to include business process models into the instructions for the stuff and to use them as essential part of the requirement specifications for the information systems. In the same time, as the numbers of diagrams is constantly growing, and the diagrams are independent VISIO files, it becomes harder and harder to keep them consistent and to avoid discrepancies. So SSIA is looking for a ways to improve technology of business process modeling.

2.2 Overview of Language ProMod

ProMod is based on a subset of BPMN and keeps its more frequently used graphical symbol: activities, events, sequence and message flows, data objects etc. These symbols sometimes have specific semantics for SSIA. For example, occurrence of an event means, that a person or an organization has brought a package of documents – events are also color-coded to show whether they originate within SSIA or come from another institution. Message and sequence flows always carry documents and are marked as significant (bold arrows) or insignificant, and so on.

Graphical symbols have rich set of attributes. Activity, for example, in addition to traditional attributes (name, textual description, performer…) has also domain-specific attributes (regulations defining it, document templates involved, customer services fulfilled…) and modeling process attributes (acceptance status, error flags, version…).

A model in ProMod is a set of diagrams representing business processes (Figure 1). There are three types of diagrams (all behavioral according to UML taxonomy):

1. Business process diagrams are used to describe business processes for employees of the organization. They are simple and easy to read.
2. Information system diagrams are also used for process modeling, but are intended to describe the process in a way, suitable for development of information systems. They contain all necessary details, but tend to be complicated. For further differentiation between these two types of diagrams see Chapter 2.6.
3. Customer service diagrams provide description of the business processes from the viewpoint of services provided to the customers – see Chapter 2.5.

Besides these diagrams, structured lists are also part of the model, representing:

1. Structure of organization,
2. Regulations and local instructions defining the business processes,
3. Information artifacts – documents and pieces of information,
4. Services provided to customers.

These lists are not only convenient way to enter values of attributes – they by themselves carry essential data, establish connection between various fragments of model and are used for consistency checking and reporting.

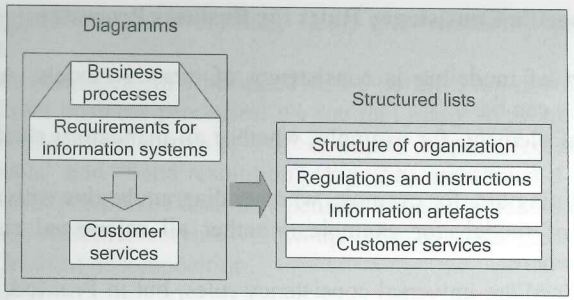

Fig. 1. Diagrams and structured lists

2.3 Refinement of Business Process Models with Enterprise Information

Information stored in the structured lists is needed not only for modeling of business processes – in fact it is the basic enterprise information. In most organizations this information can be found in some information systems, but unfortunately these systems have not been built with business process modeling in mind – they perform other functions. Traditional modeling tools often are incapable to connect to these data bases and obtain the data. Models, therefore, remain isolated from the real world, and if, for example, enterprise information is changed, it is easy to forget to change the models accordingly.

In domain-specific modeling tools it is natural to provide means for data exchange with enterprise information systems. ProMod provides these means, giving so the following advantages:

- Enterprise information can easier be maintained in information systems specially designed for it – there the information is connected to another data, quality can be checked and responsibility for maintenance can be assigned.
- Information is not duplicated, if modelling tools take it from information systems.
- Business models are closely connected to real life of the organization, and the risk for them to become outdated and inadequate is smaller.
- If needed, it is possible to integrate business process models into information systems, to show step-by-step progress of the instance of business process.
- It is possible to analyze business processes in context of enterprise data, showing, for example, which other regulations and which steps of business processes will be affected, if some part of regulation is changed (that, by the way, is especially important in SSIA, because of frequent and voluntary changes in regulations).

In addition it is possible to interconnect different diagrams and graphical symbols and to make new types of diagrams according the domain-specific logic. Customer service diagrams, for example, consists of action symbols, defined in other diagrams, and joined together, because they are needed for particular customer service (as mentioned in Chapter 2.2, customer services are part of enterprise data).

2.4 Domain-specific Consistency Rules for Business Processes

Important aspect of modeling is consistency of created models. According to the scope consistency can be:

1. In the level of element, for example, whether all mandatory attributes have been entered.
2. In the level of diagram, for example, whether diagram begins with an event.
3. In the level of model, for example, whether all referenced sub-processes are defined somewhere.

Above mentioned are universal consistency rules, but in ProMod, rules, reflecting specificity of SSIA are more essential. As the main goal of business processes is to process customer documents, it must be checked, whether all documents, provided by customer, are used in some step. It must be checked, whether set of documents from event starting the business process match to set of documents used in decomposition of its first step. It must be checked, whether all steps in all business processes are needed for some customer services, and so on.

As enterprise data is linked to the model, it is possible to check, whether performer of the step still exists in SSIA, whether customer service is still provided, whether organizational structure of SSIA have not been changed, whether regulations, defining business process, have not been expired… As the possibilities of domain-specific consistency checking seem to be unlimited, ProMod provides means for easily adding new rules.

In ProMod consistency checking is not performed during creation or modification of the diagrams – consistency check is a separate action, and inconsistent models can be stored in the system and kept for a while. This approach gives some benefits:

1. It is possible to start modeling from rough sketches made by insurance professionals of SSIA, work with them and in the end turn them into consistent models.
2. This approach is more comfortable for non-IT professionals, because they tend to concentrate on the main ideas and think, that consistency details are boring.

2.5 Business Processes and Customer Services: Two Views on the Same Model

At the very first steps of the business process modeling in organization like SSIA, when trying to identify and name business processes, there are two essentially different options. First, we can look at the organization from management's perspective and classify processes according to the way they are performed. Second, we can take perspective of customers and classify processes according to services or products they are producing (value added).

Management's perspective seems more natural, and has been chosen in SSIA (also risk of "silo thinking" is present – see [16]). We believe that it will be the case in most government institutions. If one reads statutes of SSIA, the first higher level business processes are obviously the functions mentioned there: grant social insurance services, provide consultations, register socially insured persons, etc. (see ProMod business process diagram in Fig. 2.a). Customer's perspective would lead to business processes related to the customer services: grant retirement pension (including

consultations, registration and everything else), grant disability pension or grant childbirth benefit.

If, following the management's perspective, we perform top-down decomposition of high-level abstract business processes; we see that many steps are independent of the customer services they provide. For example, steps "Receive documents", "Register documents" and "Send resolution" (Fig. 2.b) are almost the same whether request for retirement pension or disability pension is being processed. Differences in processing various customer services show up only in some steps (Fig. 2.c) and mostly in deeper level of decomposition.

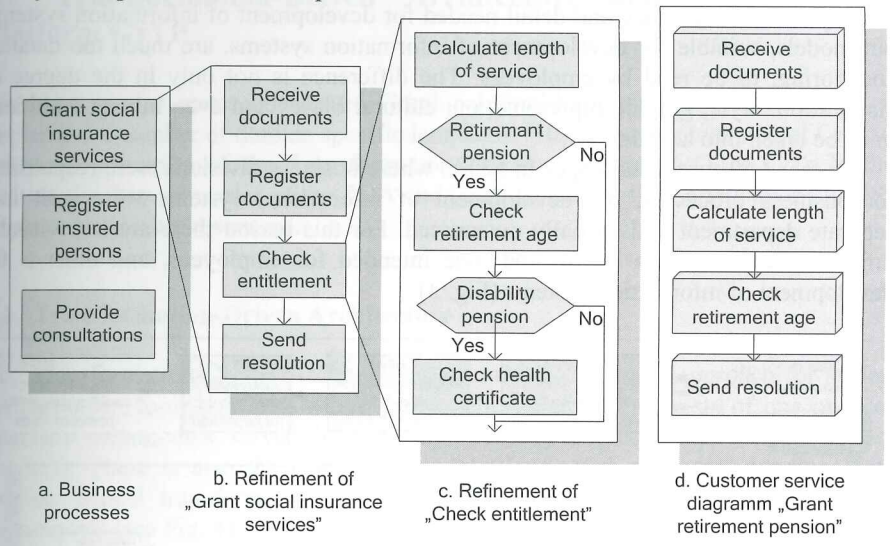

Fig. 2. Business processes and customer services

Essential drawback in taking management's perspective is that in the resulting models customer services are not clearly represented – they are "dissolved" in detailed lover-level diagrams. Customer, for example, is always interested in one service at a time and business process diagrams are not helpful to him.

To resolve this contradiction between management's and customer's perspectives, in ProMod we have introduced special type of diagrams, called Customer Service Diagrams. The Customer Service Diagram is made for every service provided, and contains all steps from various business process diagrams, needed to provide that service (Fig. 2.d). This is a distilled value chain for one particular customer service. These diagrams do not contain any new information they are just a different views to business process diagrams, and in our editor they are made semi-automatically. Customer Service Diagrams bear some resemblance to use-cases and communication diagrams in UML.

2.6 Modeling for Humans and Modeling for Information Systems

The MDA approach encourages automatic translation of business process models into implementation artifacts (representation of the business process as a database objects or executable code). In the situation when both restructuring of the business operations and development of an information system are the goals of the business process modeling, it is tempting to assume, that the same set models can be used for both purposes. This point seems even stronger, because the same modeling language can be used to pursuit both goals. However models, that can be easily read by employees, lack accuracy and detail needed for development of information systems, but models, suitable for development of information systems, are much too detailed and boring, to be read by employees. The difference is not only in the degree of elaboration: style, graphic representation, cultural biases and even human ambitions must be taken into account.

We faced all these challenges in SSIA, where business divisions were responsible for business processes, but development of information systems were split into separate department and partially outsourced. For this reason there are two visually different diagram types in ProMod: one intended for employees, and other – for development of information systems (Fig. 3).

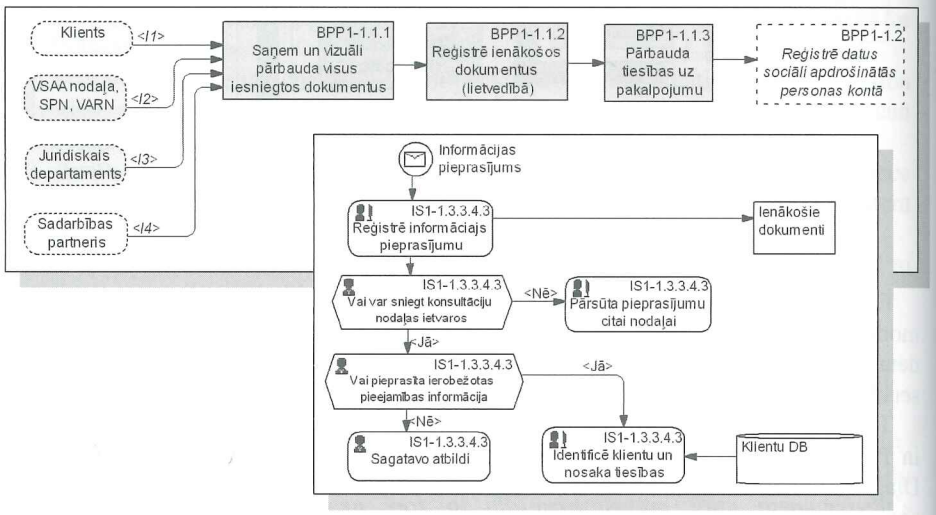

Fig. 3. Diagrams for employees and for development of information system (real-life example)

Diagrams for employees have limited set of graphical symbols; they are intended to specify sequence of steps, rather than detailed logic; and, in order not to disturb employees of business divisions, they look much like previously used IDEF0 specifications. Level of detail is acceptable, if knowledgeable insurance professionals have no difficulties to follow the business process, using them.

Diagrams for development of information systems have richer set of graphical symbols. Level of detail is higher – professional information system designers must have no difficulties to design information system using these diagrams.

Both types of diagrams are elaborated by top-down decomposition, and higher level information system diagrams in most cases are subordinated to the lower-level business diagrams.

3 Transformation-Driven Architecture and Tool Building Platform GrTP

We have used a metamodel-based Graphical Tool-building Platform GrTP [9] to implement a number of domain specific languages [10]. The recent version of GrTP is based on principles of the Transformation-Driven Architecture (TDA [7]). In this Section, the key principles of the TDA and GrTP as well as their applications in DSL implementation are discussed.

3.1 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). 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. 4).

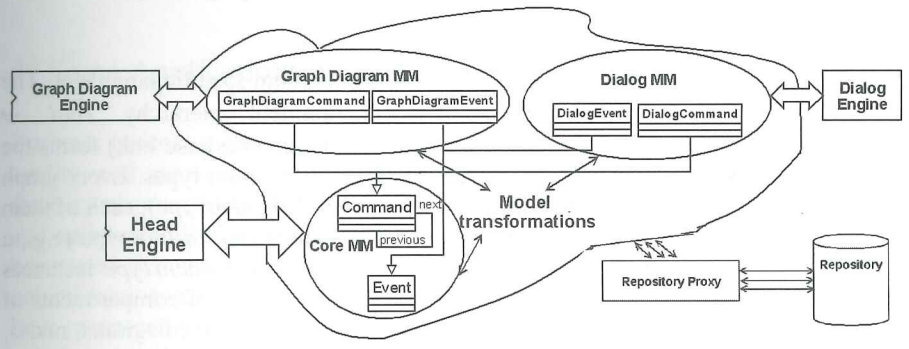

Fig. 4. Transformation-driven architecture framework filled with some interfaces

Each engine may generate events, and the corresponding transformation, being able to handle this event, is executed. Before calling the transformation, the engine in which the event occurs creates an instance of the corresponding "Event" subclass. The properties (attributes and links) of this instance may be considered as arguments for the transformation. While events are used to call transformations from engines, commands are used for the opposite direction - to call engines from transformations.

When there is a need to call an engine, the transformation creates a command and asks the Head engine to execute this command. The head engine determines which of the engines must be called and passes the control to it. Each command is an instance of some "Command" subclass.

3.2 TDA-based Tool Building Platform GrTP

Using TDA approach, 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, two basic 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 [7] embodies advanced graph drawing and layouting algorithms ([17]) as well as effective internal diagram representation structures allowing one to handle the visualization tasks efficiently even for large diagrams.
- The property dialog interface allows user to communicate with the repository using visual dialog windows.

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. 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.

3.3 Tool Definition Metamodel

First of all, let us explain the way of coding models in domain specific languages. The main idea is depicted in Fig. 5. The containment hierarchy *Tool* → *GraphDiagramType* → *ElementType* → *CompartmentType* (via base link) forms the backbone of TDMM. Every tool can serve several graph diagram types. Every graph diagram type contains several element types (instances of *ElementType*), each of them being either a box type (e.g., an *Action* in the activity diagram), or line type (e.g., a *Flow*). Every element type has an ordered collection of *CompartmentType* instances attached via its base link. These instances form the list of types of compartments of the diagram elements of this type. At runtime, each visual element (diagrams, nodes, edges, compartments) is attached to exactly one type instance.

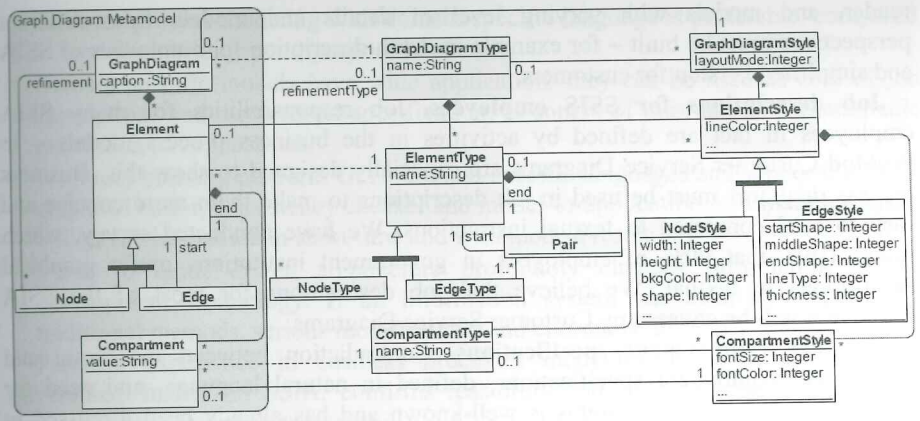

Fig. 5. The way of coding models

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 "elementCreated"(attribute of ElementType) 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.

4 Some Applications of Domain-specific models

Business process modeling is not an end in itself – models are built to make high-quality and convenient information systems. Sensitivity of social security and frequent changes in regulations (see Chapter 2.1) require high reliability, flexibility and maintainability of software. Traditional method of software developments have been used for years and have not yielded desired results. Using specifications in natural language, it was impossible to achieve needed accuracy and unambiguity. We have proposed modeling with domain-specific graphical language ProMod as a solution. Applications of the modeling that are the most urgent in SSIA are described below.

Availability of models to the wide spectrum of users. Concise description of business processes in graphical diagrams can be used as instructions for employees providing customer services and as information for clients, showing what will be done in SSIA, in order to serve their requests. The best way to spread the models is to make them available on the internet.

ProMod can export diagrams, corresponding information from structured lists and descriptive documents to Web pages. Thought not every diagram is suitable for every

reader, and models with varying level of details and models from different perspectives must be built – for example in-depth description for employees of SSIA and simplified version for customers.

Job descriptions for SSIS employees. Job responsibilities for many SSIA employees in fact are defined by activities in the business process models – in ProMod Customer Service Diagrams are especially designed to show this. Business process diagrams must be used in job descriptions to make them more concise and easy to read compared to textual instructions. We have conducted survey, which shows ([6]), that 90% of employees in government institutions prefer graphical descriptions to textual. We believe that job descriptions for most of the SSIA employees will be covered by Customer Service Diagrams.

Software requirement specifications. Contradiction between inaccurate and changeable requirement specifications, defined in natural language, and need for high-quality information systems is well-known and has already been discussed in this paper. We believe that domain specific business modeling (especially, using ProMod Information System Diagrams) will largely improve situation in SSIA.

Conversion from models to applications. We believe that approach: "Less technical programming, more concise specifications", and development of information systems without technical programming can become possible in the nearest future. Modeling is the first and mandatory stage in this process. In order to use information from business models in applications (no matter, whether they are generated from models or coded manually), it is necessary to transfer this information automatically or manually from repository of the modeling tool into application database. Manual transferring involves re-entering information about objects and their connections, and linking this information to the application data. This is a monotone and quite error-prone job. Transferring information with automated tools would be more efficient. This kind of transformation can be implemented with minor resources, if the modeling tool provides application programming interface to access its repository. So application can work according to models created in graphical language, but its quality (usability, reliability, security, performance etc.) remains independent of capacity of some hypothetic generator to generate a high-quality applications.

This approach has been tested in a number of medium-size projects [6], where information systems are less complex than in SSIA. In SSIA this is a next step to be taken. This approach has proved noticeable viability, and attitude of users towards the graphical models as requirement specifications and as core of user guides was surprisingly positive. Users considered graphical diagrams as highly comprehensible and soon gave up reading thick and boring manuals. Admittedly this approach asks for further development, but we see this as a realistic way to develop user-friendly, flexible and reliable information systems.

5 Conclusions

The following conclusions can be made from our experience with creating domain specific language ProMod and with business process modeling in SSIA:

- Business process modeling with domain-specific language is preferable, compared to modelling with general-purpose language.
- Domain specific models have wide application: they can be used as core of job descriptions and requirement specification, as source of information for automatic generation of applications.
- With tool building platform GrTP domain-specific languages and supporting tools: graphical editor, consistency checker and model-to-application information transfer utility, can be created in short time and with modest resources.
- Move to model-driven architecture profoundly changes information system development technology. If an information system has been developed with traditional methods, serious modifications and enormous resources can be needed.

Practical experience in business processes modelling in large and complex government institution SSIA, confirms feasibility and advantages of model-driven development of information systems.

Acknowledgments. This research is partly supported by European Social Fund.

References

1. UML, http://www.uml.org.
2. BPMN, http://www.bpmn.org.
3. MDA Guide Version 1.0.1. OMG, http://www.omg.org/docs/omg/03-06-01.pdf.
4. Oracle Designer, http://www.oracle.com/technology/products/designer/documentation.html
5. Flore, F.: MDA: The Proof is in Automating Transformations Between Models. OptimalJ White Paper. http://www.dsic.upv.es/~einsfran/mda/modeltransformations.pdf
6. Ceriņa-Bērziņa, J., Bičevskis, J., Karnītis, Ģ.: Information systems development based on visual Domain Specific Language BiLingva. In: 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2009), Krakow, Poland (2009)
7. Barzdins, J., Cerans, K., Kozlovics, S., Rencis, E., Zarins, A.: A Graph Diagram Engine for the Transformation-Driven Architecture. In: Proceedings of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces, pp. 29-32, Sanibel Island, USA (2009)
8. Lankhorst, M., et al.: Enterprise Architecture at Work: Modelling, Communication and Analysis. Springer (2009)
9. Barzdins, J., Zarins, A., Cerans, K., Grasmanis, M., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A., Zarins, A.: Domain Specific languages for Business Process Managment: a Case Study. In: Proceedings of DSM'09 Workshop of OOPSLA 2009, Orlando, USA (2009)
10. Barzdins, J., Kozlovics, S., Rencis, E.: The Transformation- Driven Architecture. In: Proceedings of DSM'08 Workshop of OOPSLA 2008, pp. 60—63, Nashville, USA (2008)
11. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall (2004)
12. Weiklens, T.: Systems Engineering with SysML. Morgan-Kaufman OMG Press (2007)
13. White, S.A., Miers, D., Fischer, L.: BPMN Modeling and Reference Guide. Future Strategies Inc. (2008)
14. Lan Cao, Balasubramaniam Ramesh, Matti Rossi: Are Domain-Specific Models Easier to Maintain Than UML Models? In: IEEE Software, vol. 26, no. 4, pp. 19--21 (2009)
15. ICAM Architecture Part II - Volume IV - Function Modeling Manual (IDEF0). http://handle.dtic.mil/100.2/ADB0624

16. Harmon, P.: Business Process Change. Morgan Kaufmann Publishers (2007)
17. Freivalds, K., Kikusts, P.: Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. In: Proceedings of The Latvian Academy of Sciences, Section B, vol. 55, No. 1, pp. 43–51, Riga (2001)

The design of electronic service process using YAWL

Peteris Stipravietis, Maris Ziema

Riga Technical University, Faculty of Computer Science and Information Technology,
Institute of Computer Control, Automation and Computer Engineering
Meza 1/3, 3rd floor. LV-1048, Riga, Latvia
{Peteris.Stipravietis, Maris.Ziema}@zzdats.lv

Abstract. The article discusses the solution of common business process design-time problems using YAWL. The approach proposed by the authors is based on the creation of business process in the YAWL environment in order to simulate the process model which could resolve some of the design-time problems as well as provide hints to correct initial process. The article describes technique to acquire the primitive description of process from the YAWL workflow. The primitive description is represented as oriented graph and is used to transform the YAWL workflow to another hierarchic language. To create process description in BPEL, pattern recognition algorithm is used on the process primitive description. The resulting BPEL process is then evaluated, comparing it with initial YAWL workflow and trying to execute it, involving the programmer to fix something as little as possible.

Keywords: YAWL, BPEL, simulation, transformation.

1 Introduction

E-services are common in information society nowadays, and even though they tend to become more and more accessible and varied, the problems that occur during the design phase of the service remain the same. These problems include, for example, questions on how to facilitate the creation of business process to the user with no specific programming skills, how to define the process in a way that creates the process description abstract but accurate enough at the same time, how to check the created model – to determine the weaknesses, perform the measurements based tuning, and others. The solutions of these problems rely heavily on the choice of the language used to describe the process.

Existing business process modeling languages can be divided in two groups. The languages of the first group are favored by the academic community, but rarely used in real-life solutions. These languages are based on Petri nets, process algebra; they have formal semantics, which allow the validation of the models described by these languages. The languages of the second group are used in real-life projects much more than in academic researches. BPEL, BPMN are among these languages. These, so called business languages, often lack proper semantics, which could lead to debate on how to interpret the business models described by these languages. The availability of different implementations of these languages from different vendors does not

facilitate the situation either, yet they are used much more, compared to seldom used models described by academic languages. If a situation arises when business process model described by business language needs to be validated using Petri nets, one must either abandon the validation or transform the process model to another model, described in academic language, for example YAWL. The authors propose reverse approach – first, a process is created using academic language. The design problems of the process model can then be solved by mathematical means. Second, the verified and updated model is transformed to model described in business language. The advantages of the approach described follows:

- If a model is created using academic language, it is more readable and maintainable than the model, which is a transformation result itself. It is also easier to perform analysis of untransformed model, because the transformation could lose some design information.
- Model, transformed to business language, is already validated and ready to be executed. Of course, the model must be double-checked to make sure if it needs any corrections. The alternatives of the execution environment for the model are much more than the environments for academic languages; in addition to that, they have superior technical support.

The purpose of this article is to examine the aforementioned approach – can it be used during the design of simple e-service business project, check if it helps to resolve most common design-time problems; view the transformation from academic language YAWL to business language BPEL – these languages are chosen to implement the proposed approach.

2 The languages

YAWL stands for 'Yet Another Workflow Language' – the language to describe workflows. It supports non-trivial data transformations and integration of web services. YAWL is based on extended workflow nets (EWF) [1]. YAWL could be defined as Petri net, enhanced with possibility to define multiple instances, synchronizations, OR splits and joins as well as cancellation regions. The workflow definition in YAWL in hierarchically structured set of EWF nets, forming a tree. Every net consists of activities and conditions – the places and transitions of the net. Activities can be atomic or composite. In fact, every composite activity is a net itself. Every net has an input and output condition [2]. YAWL has a graphic environment in which the workflows are created and execution environment to load and run validated definitions of workflows into. YAWL-based solutions are also used in business environment, for example, YAWL4Film is a YAWL extension used in Australian Film Television and Radio School (AFTRS) [3], but extension YAWL4Health is used in Academic Medical Centre (AMC) in Netherlands [4]. YAWL is chosen as the academic language of the approach because of the following reasons:

- YAWL is based on EWF, the workflows described in YAWL can be transformed to colored Petri nets to perform simulations and formal semantic validation;

- YAWL is designed to support all the workflow patterns [5].

BPEL stands for 'Business Process Execution Language'. Processes described with BPEL uses web services to exchange information with other processes or systems. BPEL is based on XML, it has not standardized graphical notation. BPEL is chosen as the business language of the approach because of the following reasons:

- BPEL is an industry standard;
- There are many popular environments available, in which to design and run BPEL processes – Microsoft BizTalk, Oracle BPEL Process Manager and IBM WebSphere among the others.

3 The simulation

The analysis of the created business process is very important part of the design – one needs to find bottlenecks, when instance of the process or its part could use up all available resources, thus forcing other instances to wait for these resources; identify dead ends, which could lead to infinite loops and never ending process instances; find deadlocks, when process querying for the same resources effectively block each other; define fault handling and cancellation activities, which cancel all the work done by previous activities; identify reusable structures, for example, audit log activities.

Such challenges usually are overcome with the simulation. Let us inspect a certain simulation approach and see if it helps with the problems, mentioned before. The authors of the publication [6] propose simulation which uses process design data, historical data about executed process instances from audit logs and state data of the running process instances from the execution environment. Data from all three sources are combined to create simulation model – design data are used to define the structure of the simulation model, historical data define simulation parameters, state data are used to initialize the simulation model.

Altering the simulation model allows to simulate different situations, for example, to omit certain activities or divert the process flow to other execution channels. Taking into account the state data of running process instances, it is possible to render the state of the system in near future and use the information to make decisions regarding the underlying business process.

The simulation of the workflow is carried out using process data mining framework ProM [7]. To create simulation model, following steps are performed:

- Workflow design and audit log data are imported from execution environment;
- According to imported data a new YAWL workflow model is created and state date are added;
- The new model is converted to Petri net;
- Resulting Petri net is exported to simulation execution environment CPN Tools [8] as a colored Petri net.

A simple business process which provides credit card application is used as an example (Fig. 1). This process is used as an example in many solutions, related to YAWL.

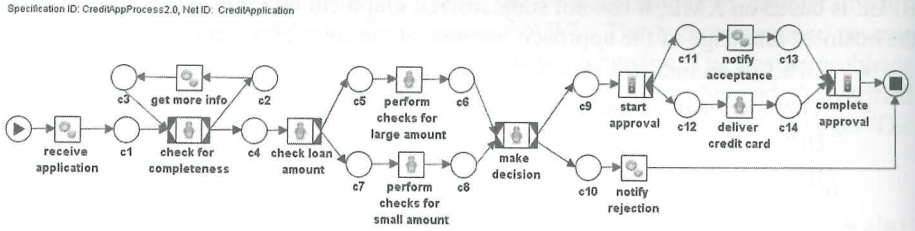

Fig. 1. YAWL workflow example [6].

Experimenting with the proposed simulation and analyzing the results, the authors of this paper conclude that this technique could be used to solve some of the designing problems mentioned before. The technique discussed differs from others (known to authors) with its degree of realism – many other methods assume that available resources are 100% dedicated to the instance being simulated and do not take into account real resource usage by other instances or processes; yet this method creates artificial delays based on historical data from audit logs. Such approach allows finding bottlenecks in the process model. The identification of dead ends is simple enough too – one has to execute the simulation using each member of the initialization parameters set and disabling delays. However, the deadlock identification is not possible, because this simulation approach does not allow multiple parallel process instances within one scope. Furthermore, the approach does not provide the cancellation simulation; hence the YAWL fault handler testing is not possible. The reason of this lack of functionality is the absence of cancellation region concept in Petri nets. Similarly, Petri nets does not support OR splits and joins (multi-choice workflow pattern, M out of N), therefore the simulation of these constructs is not possible. Such pattern is not supported by BPEL either.

4 The transformation

When the process model has been simulated and updated accordingly, it is ready to be transformed to BPEL process. Proposed transformation at first creates primitive process structure and then creates BPEL process, using pattern recognition. Similar task - the pattern recognition and transformation to BPEL - is discussed in [9] and is based on the transformation of BPMN process to Petri net and subsequent transformation of the net to BPEL process. The approach proposed by authors of this paper uses language independent primitive structure - the notation of the process flow as a directed graph preserving the semantics of the process. The algorithms described in [9] allows to transform non-well-formed BPMN processes, using the event-action mechanism of BPEL, producing usable, albeit rather unreadable BPEL. The authors of this paper tend to transform the non-well-formed primitive structure to well-

formed one (i.e. using algorithms discussed in [10]), because it would allow to transform the process to any structured language, not just BPEL.

4.1 Transition from YAWL workflow to primitive structure

To facilitate the transition from YAWL workflow to BPEL process, the bipartite graph describing the workflow is simplified, resulting in primitive process structure – oriented graph with vertices of one kind. The primitive structure is created by removing the vertices standing for YAWL net places. The condition attached to the removed place is attached to new arc between the vertices of primitive structure. There is one exception, though – the end place also is included in the primitive structure.

Let us define the YAWL workflow as (P, T, W), where:
$\mathbf{P}$ – finite set of places un $\mathbf{T}$ – finite set of transitions, $\mathbf{P} \cap \mathbf{T} = \emptyset$; but $\mathbf{W} \subseteq (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P})$ is set of arcs between places and transitions in such a way that no transition is connected with another transition and no place is connected with another place. $\mathbf{P_b} \subset \mathbf{P}$ is a subset of $\mathbf{P}$ and denotes end places of the workflow: $\mathbf{W} \cap (\mathbf{P_b} \times \mathbf{T}) = \emptyset$. The example of simple workflow is shown in Fig. 2.

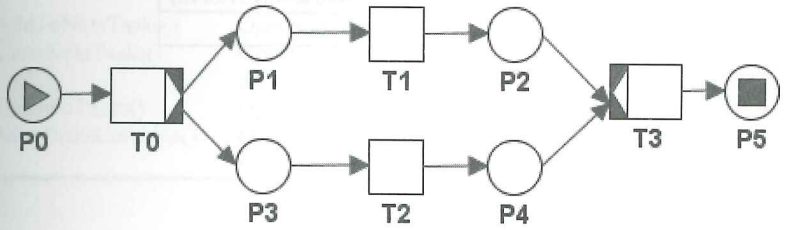

Fig. 2. Simple YAWL workflow

Let us define primitive structure (A, W), where:
A – finite set of activities, containing all elements from the workflow transition set T and all elements from the end places set P_b: $A = (T \cup P_b)$.

If $T_0 \overset{cond}{\Longrightarrow} P_n \Rightarrow T_1$ denotes a process flow from transition T_0 to transition T_1 through place P_n, when condition $cond$ allows it, then corresponding transition from A_0 to A_1 is defined in primitive structure – $A_0 \overset{cond}{\Longrightarrow} A_1$.

If $T_m \overset{cond}{\Longrightarrow} P_n$ denotes a process flow from transition T_m to place P_n, when condition $cond$ allows it and $P_n \in P_b$, then corresponding transition from A_m to A_n is defined in primitive structure – $A_m \overset{cond}{\Longrightarrow} A_n$. The example of primitive structure is shown in Fig. 3.

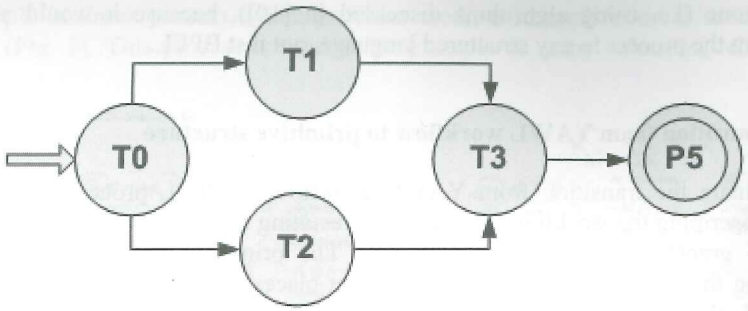

Fig. 3. Primitive structure corresponding to sample YAWL workflow

The transition from YAWL workflow to primitive structure is accomplished by analyzing the YAWL workflow description in XML. Fig. 4 shows the class diagram of primitive structure. Tables 1 and 2 describes the attributes and operations of the Process and Activity classes respectively.

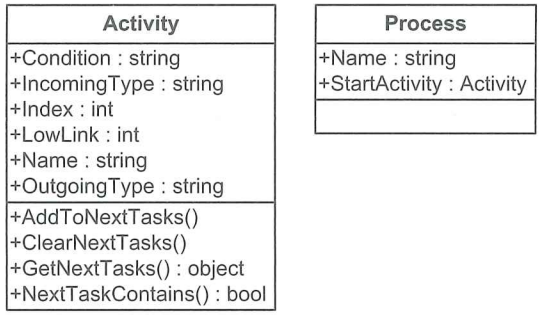

Fig. 4. Class diagram of primitive structure

Table 1. Attributes and operations of Process class.

Attribute/operation	Description
Name	The name of the primitive structure
StartActivity	The starting vertex of primitive structure

Table 2. Attributes and operations of Activity class.

Attribute/operation	Description
Condition	The condition which allows the process flow through this activity
IncomingType	Type of incoming arcs: • None – no incoming arcs (for example, start activity) • Single – one incoming arc • And – many incoming arcs, synchronizing parallel flow • Xor – many incoming arcs, synchronizing exclusive flow
Index	Index – is used for cycle identification with Tarjan SCC algorithm.
LowLink	Link – is used for cycle identification with Tarjan SCC algorithm
Name	The name of activity
OutgoingType	Type of outgoing arcs: • None – no outgoing arcs (for example, process end) • Single – one outgoing arc • And – many outgoing arcs, starting parallel flow • Xor – many outgoing arcs, starting exclusive flow
AddToNextTasks()	Operation adds new neighbor to the neighbor set
ClearNextTasks()	Operation clears neighbor set and deletes all outgoing arcs
GetNextTasks()	Operation returns neighbor set
NextTasksContains()	Operation checks if neighbor set contains given neighbor

4.2 Transition from primitive structure to BPEL process

A BPEL process is hierarchically structured activity set of sequential execution. It does not allow arbitrary cycles or goto-like constructions – the process constructions are either sequential or inclusive. It means that situation when a construction A, let us say, 'while', opens and then construction B ('flow') opens, but then A is closed and B remains open, is not valid. The opening and closing of construction blocks should follow the LIFO principle.

There are cases when vertices of primitive structure do not follow that principle – that happens when both incoming and outgoing arc count is greater than 1, i.e., the vertex both synchronizes and splits the process flow. To avoid such problems corresponding vertex is divided in two vertices – the first for synchronizing the flow but the second for splitting it. The division of vertices helps to identify patterns correctly. It also facilitates the finding of the starting and ending activities of each pattern. Fig. 5 shows a fragment of workflow where a transition synchronizes the flow and immediately splits it.

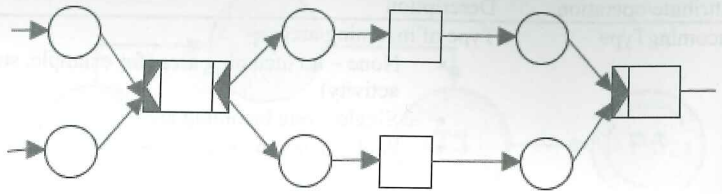

Fig. 5. The synchronization and splitting of the process flow in YAWL

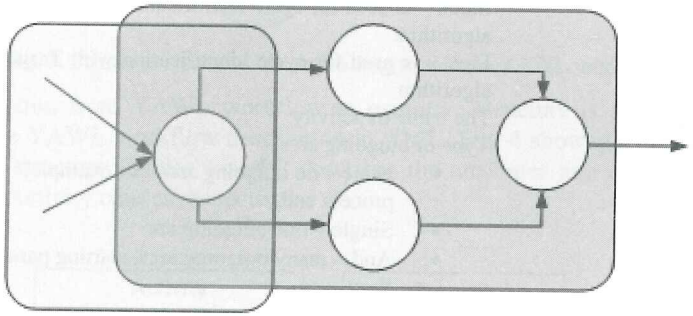

Fig. 6. Pattern overlapping in primitive structure

Fig. 6 shows overlapping of the patterns in primitive structure. The division is carried out by traversing all the vertices of primitive structure and dividing whose both incoming and outgoing arc count is greater than 1. The original vertex preserves its incoming (synchronizing) arcs, while the outgoing (splitting) arcs are added to new vertex. Both vertices are connected with new arc.

If $I(A)$ – set of incoming arcs of vertex A, $O(A)$ – set of outgoing arcs of vertex A, but $|I(A)| > 1 \wedge |O(A)| > 1$, then define vertex A':

- $O(A') = O(A)$ – copy the outgoing arcs of 'old' vertex to 'new' vertex;
- $O(A) = I(A') = \{A \Rightarrow A'\}$ – replace the outgoing arcs of 'old' vertex with a single arc to 'new' vertex, which also forms the set of incoming arcs for 'new' vertex.

The primitive structure with divided vertices and clearly separated patterns is shown in Fig. 7.

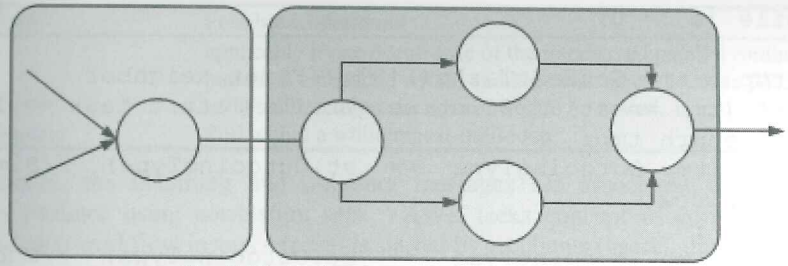

Fig. 7. Divided vertex to avoid overlapping

After the primitive structure has been updated, pattern identification in it can be started. Most common patterns are shown in Table 3.

Table 3. Most common patterns

Pattern	BPEL element	Description
Simple flow	Sequence	Allows defining a set of activities that will be invoked in an ordered sequence
Loop	While, Repeat/Until	Repeating activities
Parallel flow	Flow	Allows defining a set of activities that will be invoked in parallel. The Flow is considered finished when all parallel flows within are finished.
Exclusive choice	If	Case construct for branching the flow.

The primitive structure is processed beginning with its starting activity. This activity is transformed to BPEL Receive activity, immediately followed by Reply activity. If the count of outgoing arcs of starting activity is equal to 1, the Receive-Reply pair is enclosed by Sequence Activity. If there is more than one outgoing arc, the pair is put within Pick activity. After the starting activity algorithm recursively processes its neighbors – firstly it traverses the primitive structure using depth-first search until the end activity of initially identified pattern is found, then traverses the next neighbor of initial activity using breadth-first search. When all neighbors and the activity block have been traversed and mapped onto BPEL process, algorithm continues with next block. To find the end activity of a pattern following algorithm is used – iterate through first neighbor of each activity in loop and check if its incoming arc type is equal to outgoing arc type of initial activity. To avoid mistakes when there is, for example, another If construction within initial If construction, counter is introduced – it increases with each nested construction and decreases when nested construction is closed. The end activity of the block is found when counter becomes equal to 0.

Method to find the ending activity of the block

```
int c = 1;
Activity st = BLOCK_START_TASK;
Activity tmp = BLOCK_START_TASK;
```

```
while (c != 0)
{
  tmp = tmp.GetNextTasks()[0];//First neighbor
  if (tmp == st) //First neighbor == start task => LOOP
    return tmp;
  if (tmp.OutgoingType == st.OutgoingType) //Similar
nested block
    c++;
  if (tmp.IncomingType == st.OutgoingType) //End of
block (nested/orig)
    c--;
}
return tmp; //Block ending activity
```

Loops within the primitive structure are identified using Tarjan SCC algorithm [11]. The result is a list of cycles, where cycles are implemented as a list of activities. Looping activities are processed like other activities – every neighbor of starting activity by depth-first search until the block end activity, then breadth-first remaining neighbors.

4.3 Requirements to the YAWL workflow

To be able to transform the YAWL workflow successfully, the workflow must conform to some requirements. Firstly, it should not contain patterns, which have no analog constructions in BPEL, for example, the passing of process control to an activity residing outside the synchronized block, i.e. – goto-like construction. BPEL directly supports 13 patterns out of 20, discussed in [12]. Table 4 lists unsupported patterns and possible workarounds.

Table 4. Problematic patterns

Pattern	Possible workaround
Multi-merge	BPEL offers no support for Multi-merge pattern, as it does not allow for two active threads following the same path without creating new instances of another process
Discriminator	BPEL offers no direct support for Discriminator pattern; there are no structured activities which can be used for implementing it
Arbitrary cycles	Only structured loops like while and repeat-until are allowed. There are no goto-like constructs in BPEL.
Multiple instances with runtime knowledge	A pick activity within a while loop is used, enabling repetitive processing, triggered by three different messages: one indicating that a new instance is required, one indicating the completion of a previously initiated instance, and one indicating that no more instances need to be created [12].
Multiple instances without runtime knowledge	
Interleaved parallel routing	Multiple scopes within a flow construct that complete for a single shared variable whose access is serialized. The order of the scopes is arbitrary but serial [13]. This solution is not

Pattern	Possible workaround
	applicable if one occurrence of the interleaved parallel routing pattern is embedded within another occurrence, because BPEL serializable scopes are not allowed to be nested.
Milestone	Poll within a while/repeat-until loop.

Secondly, the incoming and outgoing messages are associated with specific process instance using correlation sets. YAWL lacks concept of correlation sets, because each workflow instance (case) is started by its clients (users), thus creating an instance in execution environment. This environment manages the workflows and offers to users corresponding options, based on the state of instance and its specification [14]. The variable which could be used as an correlation set variable must be created in the workflow or during the transformation and finally added to each defined data type used in BPEL messages.

Thirdly, support of human tasks – all BPEL activities related to exchange of information with process partners are perceived as web service operations, i.e., BPEL has no concept of "Human interaction". To fill this gap several BPEL extensions are proposed, for example, BPEL4People [15] – OASIS is working on standardizing this extension, while IBM offers implementation in its WebSphere environment [16].

Last but not least, workflow definition must correctly define all the branching conditions, lest transformed BPEL process' While, Repeat/Until and If blocks contain incorrect values.

4.4 Example

For an example authors use the same business process which provides credit card application and was used as a simulation model (Fig. 1). The process consists of four blocks – receive application, check data completeness and ask for more if necessary, make decision and inform the client. This example contains all most common patterns – simple task, parallel flow, loop and exclusive choice. Transforming the workflow to primitive structure, tasks "Check for completeness" and "Make decision" are divided to avoid pattern overlapping. Fig. 8 shows the primitive structure of example workflow with already divided vertices and identified BPEL patterns.

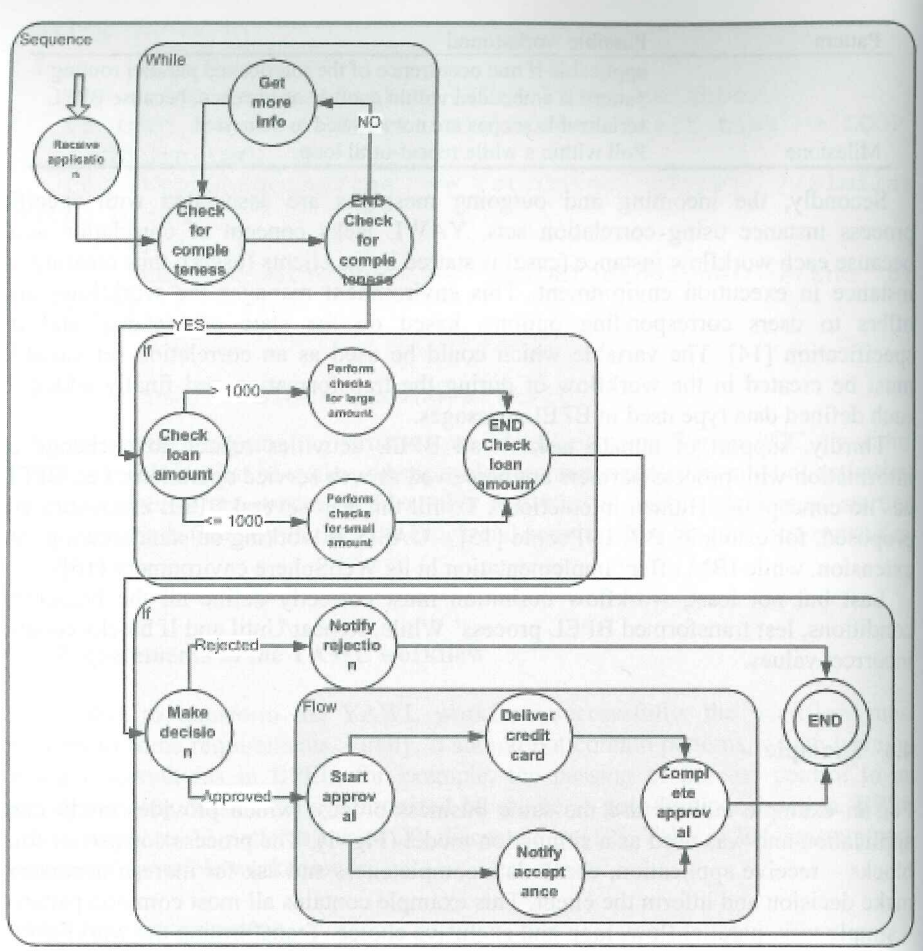

Fig. 8. The primitive structure of credit application workflow

After the creation of primitive structure BPEL process description is generated and is shown below.

```
<process name="CreditApp" targetNamespace=http://example.com/bpel
xmlns=http://docs.oasis-open.org/wsbpel/2.0/process/executable
xmlns:sref="http://docs.oasis-open.org/wsbpel/2.0/serviceref">
  <sequence>
    <receive name="receive_application_3" createInstance="yes"
      operation="op__receive_application_3"
      partnerLink="pl__receive_application_3"/>
    <reply name="receive_application_3"
      operation="op__receive_application_3"
      partnerLink="pl__receive_application_3"/>
    <while name="check_for_completeness_9_part01">
      <condition>/CreditApp/isEnoughInfo/text() = 'false'</condition>
      <invoke name="get_more_info_8" operation="op__get_more_info_8"
        partnerLink="pl__get_more_info_8"/>
    </while>
    <if name="check_loan_amount_10">
```



```
      <condition> number(/CreditApp/amount/text()) < 1001</condition>
      <sequence>
        <invoke name="perform_checks_for_small_amount_14"
          operation="op__perform_checks_for_small_amount_14"
          partnerLink="pl__perform_checks_for_small_amount_14"/>
      </sequence>
      <elseif>
       <condition>number(/CreditApp/amount/text()) > 1000</condition>
        <sequence>
          <invoke name="perform_checks_for_large_amount_13"
            operation="op__perform_checks_for_large_amount_13"
            partnerLink="pl__perform_checks_for_large_amount_13"/>
        </sequence>
      </elseif>
    </if>
    <invoke name="make_decision_17_part01"
      operation="op__make_decision_17_part01"
      partnerLink="pl__make_decision_17_part01"/>
    <if name="make_decision_17_part02">
      <condition>/CreditApp/isAccepted/text() = 'false'</condition>
      <sequence>
        <invoke name="notify_rejection_20"
          operation="op__notify_rejection_20"
          partnerLink="pl__notify_rejection_20"/>
      </sequence>
      <elseif>
        <condition>/CreditApp/isAccepted/text() = 'true'</condition>
        <sequence>
          <flow name="start_approval_21">
            <sequence>
              <invoke name="notify_acceptance_25"
                operation="op__notify_acceptance_25"
                partnerLink="pl__notify_acceptance_25"/>
            </sequence>
            <sequence>
              <invoke name="deliver_credit_card_27"
                operation="op__deliver_credit_card_27"
                partnerLink="pl__deliver_credit_card_27"/>
            </sequence>
          </flow>
          <invoke name="complete_approval_24"
            operation="op__complete_approval_24"
            partnerLink="pl__complete_approval_24"/>
        </sequence>
      </elseif>
    </if>
  </sequence>
</process>
```

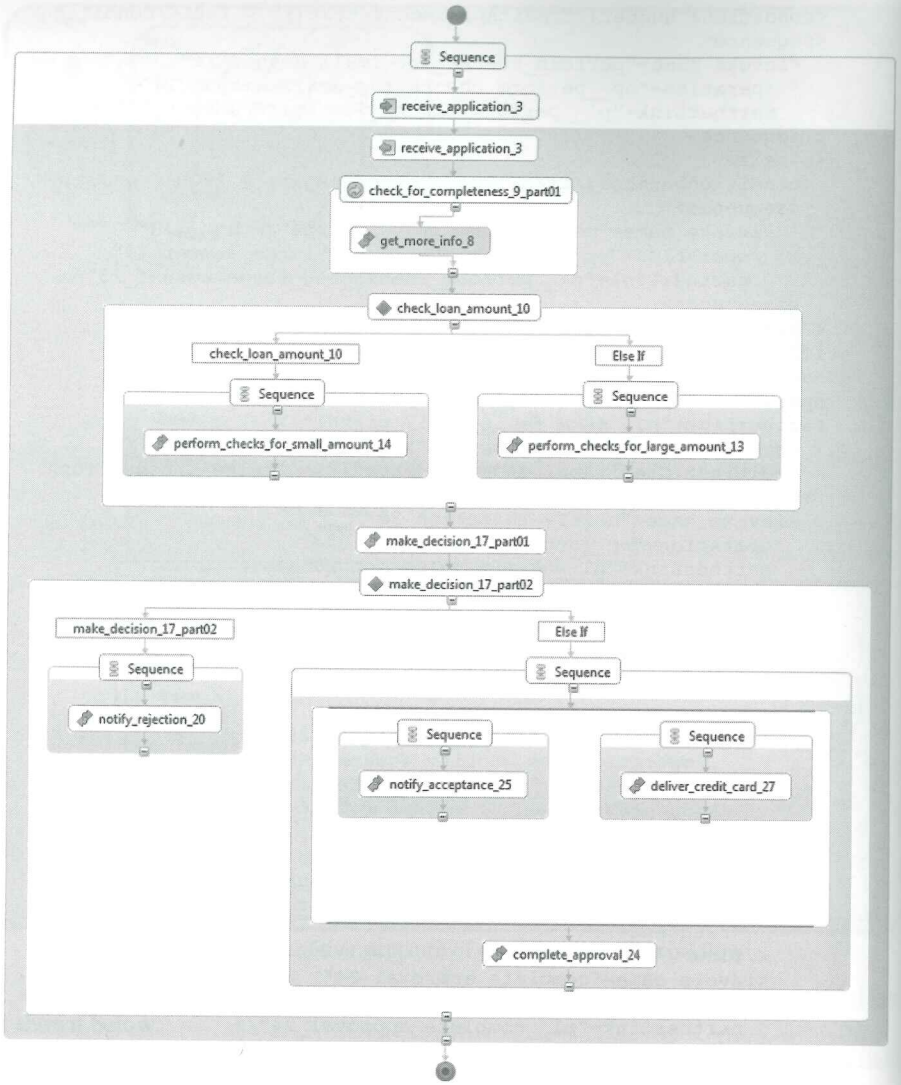

Fig. 9. Visualization of BPEL process

Fig. 9 shows generated BPEL process. The process structure is transformed correctly; however, analyzing the XML version reveals that some important blocks of process definition are absent:

- partnerLinks, which define the links between process activities and web service WSDLs;
- variables, which describe the variables and their data types, used in process. The transformation does not support the creation of this block as well as partnerLinks yet, although it is possible, because YAWL workflow definition also contains description of variables, data types and invoked web service endpoints and operations;
- faultHandlers, which describe exception handling and cancellation;
- correlationSets, which define correlation sets used in process binding with messages.

5 Conclusion

The proposed approach of business process modeling, at first creating the model in academic language to be able to perform mathematically based analysis; and then transforming it to the process described in business language with wider supported tool choice, is quite successful. The main benefit of this approach is creation of primitive structure and traversal and pattern identifying algorithm, which allows the transformation from YAWL workflow to any other hierarchical language, both academic and business. The traversal and identifying algorithm is applicable to the processes defined in other languages; however, these processes must be altered correspondingly to avoid the pattern overlapping problem.

The simulation model has some flaws – simpler design problems, for example, bottleneck and dead end identification can be achieved with this approach; however, more complex problems, such as deadlock identification or the operation of cancellation region, could not be resolved. To be honest, it has more to do with Petri nets, used in the simulation model, because these do not support multiple process instances or cancellation regions. It is possible that using the simulation model based on other mathematical framework, these problems could be solved.

The proposed transformation successfully recognized the patterns used in YAWL workflow and rendered the structure of the process; however, it missed some important process description blocks. Some of them could not be created using the proposed approach, such as faultHandlers, because simulation model used, based on Petri nets, does not support the simulation of exception handling. Some blocks simply were not processed – variables and partnerLinks fall into this category; while some were not possible to create at all, like correlationSets. The example used did not contain more complicated patterns, such as Foreach.

Summarizing the results, authors conclude that approach proposed in this article must be developed further. The main challenges are – the transformation must be able to create primitive structure from more complicated workflow, where patterns such as Foreach or arbitrary loops are used, at the same time not forgetting about pattern overlapping problem. Other simulation models must be examined to see if they could resolve the problems proved too hard for model used in this article, but transformation model has to be able to create variable and partnerLinks blocks. Last but not least, the requirements against the YAWL workflow model must be defined, so it could be possible to automatically create both faultHandlers and correlationSets blocks.

References

1. M. Weske "Business Process Management", Springer 2007, p. 169
2. W. M. P. van der Aalst, A. H. M. ter Hofstede: YAWL: Yet Another Workflow Language. Inf. Syst., vol. 30(4), pp 245–275 (2005)
3. C. Ouyang, A.H.M. ter Hofstede, M. La Rosa, M. Rosemann, K. Shortland and D. Court, Camera, Set, Action: Automating Film Production via Business Process Management. In Proceedings of the International Conference "Creating Value: Between Commerce and Commons", Brisbane, Australia, 2008
4. YAWL4Health, http://www.yawlfoundation.org/casestudies/health
5. N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar: Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org (2006)
6. A. Rozinat, M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. J. Fidge: Workflow Simulation for Operational Decision Support Using Design, Historic and State Information. Proceedings of the 6th International Conference on Business Process Management (BPM 2008), 2008, Milan, Italy. LNCS, vol. 5240, pp 196 – 211. Springer (2008).
7. W.M.P. van der Aalst, B.F. van Dongen, C.W. Gunther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007), LNCS, vol. 4546, pp 484-494. Springer, Berlin (2007)
8. K. Jensen, L.M. Kristensen, and L. Wells: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer, vol. 9(3-4), pp 213-254 (2007)
9. Chun Ouyang , Marlon Dumas , Wil M. P. Van Der Aalst , Arthur H. M. Ter Hofstede , Jan Mendling, From business process models to process-oriented software systems, ACM Transactions on Software Engineering and Methodology (TOSEM), v.19 n.1, p.1-37, August 2009
10. J. Koehler and R. Hauser. Untangling unstructured cyclic flows - A solution based on continuations. In R. Meersman, Z. Tari, W.M.P. van der Aalst, C. Bussler, and A. Gal et al., editors, On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, volume 3290 of Lecture Notes in Computer Science, pages 121–138, 2004.
11. Strong Connectivity, http://www.ics.uci.edu/~eppstein/161/960220.html#sca
12. P. Wohed, W. van der Aalst, M. Dumas, A. H. M. ter Hofstede: Pattern Based Analysis of BPEL4WS. FIT Technical Report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002
13. M. Havey: Essential Business Process Modeling. p. 141, O'Reilly (2005)
14. W. M. P. van der Aalst, L. Aldred, M. Dumas, T. A. H. M. Hofstede: Design and Implementation of the YAWL System. Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04), Riga, Latvia, LNCS vol. 3084, pp 142–159, Springer (2004).
15. Holmes T., Vasko M., Dustdar S.: VieBOP: Extending BPEL Engines with BPEL4People. 16th Euromicro International Conference on Parallel, Distributed and network-based Processing 2008, 547-555. February 2008.
16. WS-BPEL Extension for People, http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/

Grammatical Aspects for Language Descriptions

Andrey Breslav*

abreslav@gmail.com
ITMO University
St. Petersburg, Russia

Abstract. For the purposes of tool development, computer languages are usually described using context-free grammars with annotations such as semantic actions or pretty-printing instructions. These descriptions are processed by generators which automatically build software, e.g., parsers, pretty-printers and editing support.
In many cases the annotations make grammars unreadable, and when generating code for several tools supporting the same language, one usually needs to duplicate the grammar in order to provide different annotations for different generators.
We present an approach to describing languages which improves readability of grammars and reduces the duplication. To achieve this we use Aspect-Oriented Programming principles. This approach has been implemented in an open-source tool named GRAMMATIC. We show how it can be used to generate pretty-printers and syntax highlighters.

1 Introduction

With the growing popularity of Domain-Specific Languages, the following types of supporting tools are created more and more frequently:

- Parsers and translators;
- Pretty-printers;
- Integrated Development Environment (IDE) add-ons for syntax highlighting, code folding and outline views.

Nowadays these types of tools are usually developed with the help of generators which accept language descriptions in the form of annotated (context-free) grammars.

For example, tools such as YACC [7] and ANTLR [12] use grammars annotated with embedded semantic actions. As an illustration of this approach first consider an annotation-free grammar for arithmetic expressions (Listing 1.1). To generate a translator, one has to annotate the grammar rules with embedded semantic actions. Listing 1.2 shows the rule `expr` from Listing 1.1 annotated for ANTLR v3.

* This work was partly done while the author was a visiting PhD student at University of Tartu, under a scholarship from European Regional Development Funds through Archimedes Foundation.

```
expr : term ((PLUS | MINUS) term)* ;
term : factor ((MULT | DIV) factor)* ;
factor : INT | '(' expr ')' ;
```

Listing 1.1. Grammar for arithmetic expressions

```
expr returns [int result] :
    t=term {result = t;}
    ({int sign = 1;} (PLUS | MINUS {sign = -1;})
                        t=term {result += sign * t;})*;
```

Listing 1.2. Annotated grammar rule

As can be seen, the context-free grammar rule is not easily readable in Listing 1.2 because of the actions' code interfering with the grammar notation. This problem is common for annotated grammars. We will refer to it as *tangled grammars*.

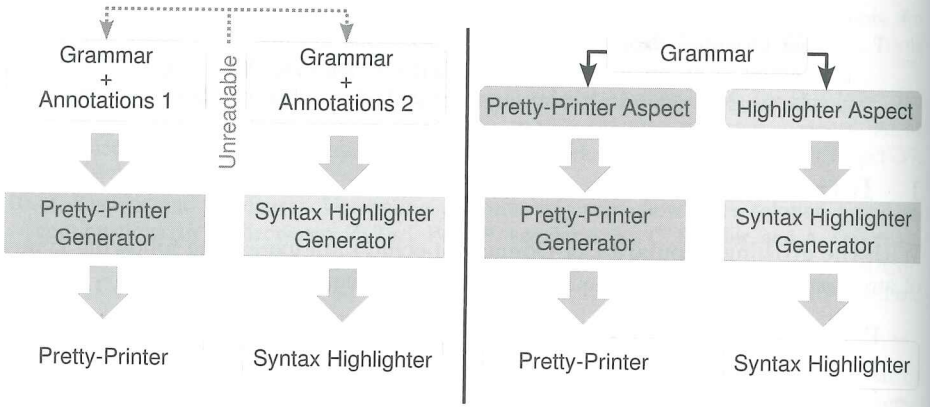

Fig. 1. Generating two supporting tools for the same language

In most applications we need to create several supporting tools for the same language (see Figure 1, left side). In such a case one uses different generators to obtain different programs (e.g., PRETZEL [3] to build a pretty-printer and XTEXT [1] to create an Eclipse editor). Each generator requires its own specific set of annotations, and the developer has to write the same grammar several times with different annotations for each generator. Besides the duplication of effort, when the language evolves, this may lead to inconsistent changes in different copies of the grammar, which may cause issues which are hard to detect. We will refer to this problem as *grammar duplication*.

This paper aims at reducing tangling and duplication in annotated grammars. A high-level view of our approach is illustrated in Figure 1 (right side): the main idea is to separate the annotations from the grammar by employing the principles similar to those behind the AspectJ language [8], this leads to a notion of a *grammatical aspect*. Our approach is implemented in an open-source tool named GRAMMATIC[1].

In Section 2 we briefly describe the main notions of aspect-oriented programming in AspectJ. An overview of grammatical aspects and related concepts is given in Section 3. Section 4 studies the applications of GRAMMATIC to generating syntax highlighters and pretty-printers on the basis of a common grammar. We analyze these applications and evaluate our approach in Section 5. Related work is described in Section 6. Section 7 summarises the contribution of the paper and introduces possible directions of the future work.

2 Background

Aspect-Oriented Programming (AOP) is a body of techniques aimed at increasing modularity in general-purpose programming languages by separating cross-cutting concerns. Our approach is inspired by AspectJ [8], an aspect-oriented extension of Java.

AspectJ allows a developer to extract the functionality that is scattered across different classes into modules called *aspects*. At compilation- or run-time this functionality is *weaved* back into the system. The places where code can be added are called *join points*. Typical examples of join points are a method entry point, an assignment to a field, a method call.

AspectJ uses *pointcuts* — special constructs that describe collections of join points to weave the same piece of code into many places. Pointcuts describe method and field signatures using patterns for names and types. For example, the following pointcut captures *calls of all public* `get-methods` *in the subclasses of the class* `Example` *which return* `int` *and have no arguments*:

```
pointcut getter() : call(public int Example+.get*())
```

The code snippets attached to a pointcut are called *advice*; they are weaved into every join point that matches the pointcut. For instance, the following advice writes a log record after every join point matched by the pointcut above:

```
after() : getter() {
    Log.write("A get method called");
}
```

In this example the pointcut is designated by its name, `getter`, that follows the keyword `after` which denotes the position for the code to be weaved into. An *aspect* is basically a unit comprising of a number of such pointcut-advice pairs.

[1] The tool is available at `http://grammatic.googlecode.com`

3 Overview of the approach

GRAMMATIC employs the principles of AOP in order to tackle the problems of tangling and duplication in annotated grammars. We will use the grammar from Listing 1.1 and the annotated rule from Listing 1.2 to illustrate how the terms such as "pointcut" and "advice" are embodied for annotated grammars.

3.1 Grammatical join points

Figure 2 shows a structured representation (a syntax diagram) of the annotated rule from Listing 1.2. It shows the annotations attached to a symbol definition

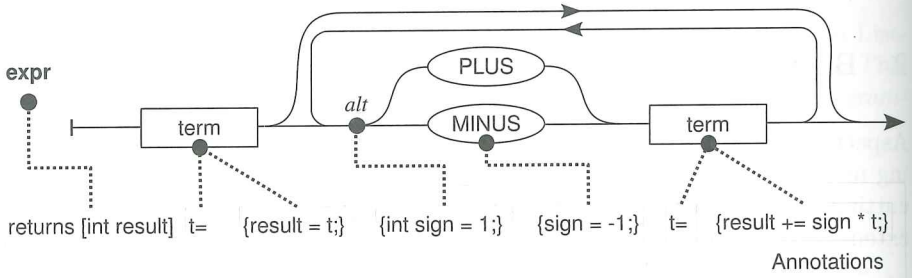

Fig. 2. Annotations attached to a grammar rule

`expr`, three symbol references: `term` (two times) and `MINUS`, and an alternative `(PLUS | MINUS)` (marked "*alt*" in the figure). All these are examples of *grammatical join points* (in Figure 2 they are marked with black circles). The full list of join point types comprises all the types of nodes of the abstract syntax trees (ASTs) of the *language of grammars*. To avoid confusion with ASTs of languages defined by the grammar, we will refer to the AST of the grammar itself as *grammar tree* (GT).

GRAMMATIC uses a notation for grammars which is based on the one used by ANTLR. The only two differences are (i) in GRAMMATIC productions are explicit and separated by ":", and (ii) an empty string is denoted explicitly by "`#empty`". Here is the list of types of GT nodes (which are also the types of the join points) with comments about the concrete syntax:

- Grammar;
- Definitions of terminal and nonterminal symbols (grammar rules) and references to them;
- Individual productions (a rule comprises one or more productions separated by ":");
- Concatenation (sequence), Alternative ("|"), Iteration ("*", "+", "?");
- Empty string ("`#empty`");


```
rulePattern
    : var? symbolPattern productionPattern* ';' ;
var
    : '$' NAME '=' ;
symbolPattern
    : '#'          // any symbol
    : NAME ;
productionPattern
    : var? ':' alternativePattern
    : ':' var? '{...}' ;
alternativePattern
    : sequencePattern ('|' (sequencePattern | (var? '...')))* ;
sequencePattern
    : iterationPattern+ ;
iterationPattern
    : var? atomicPattern ('*' | '+' | '?')? ;
atomicPattern
    : '(' alternativePattern ')'
    : symbolReferencePattern
    : '#empty'    // empty string
    : '..'        // any sequence
    : '#lex'      // any lexical literal
    : '$' NAME ; // a variable
```

Listing 1.3. Grammar of the pattern language

- Lexical literals (quoted strings).

The grammars given in this paper (e.g., Listing 1.3) may serve as example usages of this notation.

3.2 Grammatical pointcuts

GRAMMATIC implements pointcuts using *patterns* over the grammar language. A pattern is an expression that matches a set of nodes in a GT. The syntax of the pattern language is given in Listing 1.3.

The most basic form of a pattern is a direct citation from a grammar:

```
expr: term ((PLUS | MINUS) term)*;
```

This pattern matches a rule of exactly the same form (rule `expr` from Listing 1.1).

In addition to this capability the pattern language makes use of various types of wildcards which make patterns more abstract and flexible. Table 1 summarizes available wildcards and the node types they each match.

Consider some examples of patterns for rules from Listing 1.1:

- `expr : {...}` — a rule defining a symbol "expr", comprising any number of any productions (in Listing 1.1 it matches only the rule for `expr`);

Notation	Matches any...
#	Symbol
#lex	Lexical literal
..	Sequence
...	Nonempty set of alternatives
{...}	Nonempty set of productions

Table 1. Wildcards

- `# : term ..` — a production for any symbol, starting with a reference to a symbol named "term" (also matches only the rule for `expr`);
- `# : # (..)*` — a symbol reference followed by a star iterating an arbitrary sequence (matches the rules for `expr` and `term`).

The pattern language also supports variables: a part of a pattern may be associated with a name which may be used later in the same pattern, for example:

```
# : $tr=# ((PLUS | MINUS) $tr)*
```

Here the variable `$tr` is defined with the pattern # (any symbol) which means that all usages of the variable will match only occurrences of the same symbol. This pattern matches the rule for `expr` because the same symbol `term` is referenced in the positions matched by the variable `$tr`.

Note that in general a variable is bound to a *set* of GT nodes: if we match the rule for `expr` against the pattern in the example above, the variable `$tr` will be bound to a set comprised by two distinct references to the symbol `term`.

3.3 Grammatical advice

Annotations attached to grammars (they are analogous to AspectJ's *advice*) may have an arbitrarily complicated structure: in general, a generator may need a very rich annotation system. GRAMMATIC provides a *generic annotation language*, which represents the annotations as sets of name-value pairs (see Listing 1.4) which we call *attributes*. Examples of such pairs are given in Table 2 which shows all the predefined value types. Values may also have user-defined types which can be plugged into the position marked by `<additionalValueTypes>` in the grammar.

Example	Value type
int = 10	Integer
str = 'Hello'	String
id = SomeName	Name literal
rec = {b = c; d = 5}	Annotation
seq = {{1, a b 'str'}}	Sequence of values

Table 2. Predefined value types


```
annotation
    : '{' (attribute (';' attribute?)*)? '}'
    : '.' attribute ;
namespace
    : NAME ':' ;
attribute
    : namespace? NAME ('=' value)? ;
value
    : character
    : INT
    : STRING
    : NAME
    : annotation
    : '{{' (value | punctuation)* '}}'
    : <additionalValueTypes> ;
punctuation
    : '`' | '~' | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*'
    | '(' | ')' | '-' | '+' | '=' | '|' | '\\'| '[' | ']' | ';'
    | ':' | ',' | '.' | '/' | '?' | '<' | '>' ;
```

Listing 1.4. Grammar of the advice language

For example, the annotations in Figure 2 may be represented as values of type String (other representations are also possible).

As the usage of the term "attribute" may be misleading in this context, we would like to note that the approach presented here does not directly correspond to attribute grammars [9]. In fact, grammars with annotations do not have any particular execution semantics (each generator interprets the annotations in its own way), as opposed to attributed grammars which have a fixed execution semantics. One can describe attribute grammars using GRAMMATIC and define corresponding semantics in a generator, but this is just an example application.

3.4 Grammatical aspects

Now, having described all the components, we can assemble a *grammatical aspect* as a set of pointcuts-advice pairs. Usage of grammatical aspects is illustrated by Figure 1 (right side).

The syntax of grammatical aspects is given in Listing 1.5. An aspect consists of an optional *grammar annotation* and zero or more *annotation rules*. Annotation rules associate grammatical pointcuts (rule patterns) with advice (annotations). Here is an example of an annotation rule:

```
expr : $tr=# (.. $tr)*    // pointcut (pattern)
    $tr.varName = 't' ;   // advice (annotation)
```

In a simple case exemplified here, an annotation (`.varName = 't'`, the alternative syntax is `{varName = 't'}`) is attached to GT nodes to which a

```
aspect
    : grammarAnnotation? annotationRule* ;
grammarAnnotation
    : annotation ;
annotationRule
    : multiplicity? rulePattern subrules ;
subrules
    : (subpattern | variableAnnotation)* ;
subpattern
    : '@' multiplicity?  (productionPattern | alternativePattern)
        ':' (subrules | annotation) ;
variableAnnotation
    : '$' NAME annotation ;
multiplicity
    : '[' intOrInfinity ('..' intOrInfinity)? ']' ;
intOrInfinity
    : INT | '*' ;
```

Listing 1.5. Grammar of the aspect language

variable (`$tr`) is bound. For more complicated cases, one can define *subpatterns* — patterns which are matched against nodes situated under the matched one in the GT. For example, the following construct attaches an attribute named `varName` to each reference to the symbol `term` *inside a rule matched by a top-level pattern*:

```
expr : ..                    // pointuct (pattern)
    @$tr=(term):             // pointcut (subpattern)
       $tr.varName = 't' ; // advice (annotation)
```

This example illustrates the typical usage of subpatters where all annotations are associated with a variable bound to the whole pattern. As a shorthand for this situation GRAMMATIC allows to omit the variable (it will be created implicitly). Using this shorthand we can abridge the previous example to the following:

```
expr : ..
    @term: { varName = 't' } ; // '{ a = b }' equals '.a = b'
```

Note that subpatterns may have their own subpatterns.

Patterns and subpatterns may be preceded by a *multiplicity directive*, for example

```
[0..1] # : $tr=# (.. $tr)* // pointcut with multiplicity
    // some advice
```

Multiplicity determines a number of times the pattern is allowed to match. The default multiplicity is `[1..*]` which means that each pattern with no explicit multiplicity is allowed to match one or more times. When an aspect is applied to a grammar, if the actual number of matches goes beyond the range allowed by a multiplicity directive, GRAMMATIC generates an error message. In the example above, such a message will be generated for the grammar from Listing 1.1 because the pattern matches two rules: `expr` and `term`, which violates the specified multiplicity `[0..1]`.

3.5 Generation-time behaviour

Grammatical aspects are applied at generation time. Before a generator starts working, in order to prepare the data for it, GRAMMATIC performs the following steps:

- parse the `grammar` and the `aspect`
- attach the *grammar annotation* to the root node of the `grammar`
- **for each** `annotation rule` **in** `aspect`
  - **call** APPLYPATTERN(`rule pattern`, `grammar`)

Where APPLYPATTERN is a recursive subroutine defined by the following pseudocode:

APPLYPATTERN (`pattern`, `node`) **is**

- find `subnodes` matching `pattern` among descendants of `node`
  (Variable bindings are saved in `boundTo` map)
- **if** the number of `subnodes` violates `pattern.multiplicity`
  - Report **error** and stop
- **for each** `subnode` **in** `subnodes`
  - **for each** `subpattern` **in** `pattern.subpatterns`
    - **call** APPLYPATTERN(`subpattern`, `subnode`)
  - **for each** `var` **in** `pattern.variables`
    - **for each** `boundNode` **in** `boundTo(var)`
      - attach `var.annotation` to `boundNode`

end

The innermost loop goes through the set of GT nodes to which the variable `var` is bound (see Section 3) and attaches the annotations associated with this variable to each of these nodes.

If no error was reported, the resulting structure (GT nodes with attached annotations) is passed to the generator which processes it as a whole and needs no information about aspects.

Thus, GRAMMATIC works as a front-end for generators that use its API. To use a pre-existing tool, for example, ANTLR, with grammatical aspects, one can employ a small generator which calls GRAMMATIC to apply aspects to grammars, and produces annotated grammars in the ANTLR format.

4 Applications

In this section we show how one can make use of grammatical aspects when generating syntax highlighters and pretty-printers on the basis of the same grammar.

```
normalClassDeclaration
    : 'class' IDENTIFIER typeParameters?
            ('extends' type)? ('implements' typeList)? classBody ;
classBody
    : '{' classBodyDeclaration* '}' ;
typeParameters
    : '<' typeParameter (',' typeParameter)* '>' ;
typeParameter
    : IDENTIFIER ('extends' bound)? ;
bound
    : type ('&' type)* ;
type
    : IDENTIFIER typeArgs? ('.' IDENTIFIER typeArgs?)* ('[' ']')*
    : basicType ;
typeArgs
    : '<' typeArgument (',' typeArgument)* '>' ;
typeArgument
    : type
    : '?' (('extends' | 'super') type)? ;
```

Listing 1.6. Class declaration syntax in Java 5

4.1 Specifying syntax highlighters

A syntax highlighter generator creates a highlighting add-on for an IDE, such as a script for `vim` editor or a plug-in for Eclipse. For all targets the same specification language is used: we annotate a grammar with *highlighting groups* which are assigned to occurrences of terminals. Each group may have its own color attributes when displayed. Common examples of highlighting groups are *keyword*, *number*, *punctuation*.

In many cases syntax highlighters use only lexical analysis, but it is also possible to employ light-weight parsers [11]. In such a case grammatical information is essential for a definition of the highlighter. Below we develop an aspect for the Java grammar which defines groups for keywords and for *declaring occurrences* of class names and type parameters. A declaring occurrence is the first occurrence of a name in the program; all the following occurrences of that name are *references*. Consider the following example:

```
class Example<A, B extends A> implements Some<? super B>
```

This illustrates how the generated syntax highlighter should work: the declaring occurrences are underlined (occurrences of `?` are always declaring) and the keywords are shown in bold. This kind of highlighting is helpful especially while developing complicated generic signatures.

Listing 1.6 shows a fragment of the Java grammar [4] which describes class declarations and type parameters. In Listing 1.7 we provide a grammatical aspect which defines three highlighting groups: *keyword*, *classDeclaration* and *typeParameterDeclaration*, for join points inside these rules.


```
# : 'class' IDENTIFIER ..
    @#lex: { group = keyword } ;
    @IDENTIFIER: { group = classDeclaration } ;
typeParameter : IDENTIFIER ..
    @#lex: { group = keyword } ;
    @IDENTIFIER: { group = typeParameterDeclaration } ;
typeArgument : {...}
    @#lex: { group = keyword } ;
    @'?': { group = typeParameterDeclaration } ;
```

Listing 1.7. Highlighting aspect for class declarations in Java

Each annotation rule from Listing 1.7 contains two subpatterns. The first one is `#lex`: it matches every lexical literal. For example, for the first rule it matches `'class'`, `'extends'` and `'implements'`; the highlighting group `keyword` is assigned to all these literals.

The second subpattern in each annotation rule is used to set a corresponding highlighting group for a declaring occurrence: for classes and type parameters it matches `IDENTIFIER` and for wildcards — the `'?'` literal.

When the aspect is applied to the grammar, GRAMMATIC attaches the `group` attribute to the GT nodes matched by the patterns in the aspect. The obtained annotated grammar is processed by a generator which produces code for a highlighter.

4.2 Specifying pretty-printers

By applying a different aspect to the same grammar (Listing 1.6), one can specify a pretty-printer for Java. A pretty-printer generator relies on annotations describing how tokens should be aligned by inserting whitespace between them.

In Listing 1.8 these annotations are given in the form of attributes `before` and `after`, which specify whitespace to be inserted into corresponding positions. Values of the attributes are *sequences* ({{ ... }}) of strings and name literals `increaseIndent` and `decreaseIndent` which control the current level of indentation.

The most widely used values of `before` and `after` are specified in a *grammar annotation* by attributes `defaultBefore` and `defaultAfter` respectively, and not specified for each token individually. In Listing 1.8 the default formatting puts nothing before each token and a space — after each token; it applies whenever no value was set explicitly.

5 Discussion

This paper aims at coping with two problems: tangled grammars and grammar duplication. When using GRAMMATIC, a single annotated grammar is replaced

```
{ // Grammar annotation
    defaultAfter = {{ ' ' }};
    defaultBefore = {{ '' }};
}

classBody : '{' classBodyDeclaration* '}'
    @'{': { after = {{ '\n' increaseIndent }} } ;
    @classBodyDeclaration: { after = {{ '\n' }} } ;
    @'}': {
        before = {{ decreaseIndent '\n' }};
        after = {{ '\n' }};
    };
typeParameters : '<' typeParameter (',' typeParameter)* '>'
    @'<': { after = {{ '' }} } ;
    @typeParameter: { after = {{ '' }} } ;
```

Listing 1.8. Pretty-printing aspect for class declarations in Java

by a *pure* context-free grammar and a set of grammatical aspects. This means that the problem of *tangled grammars* is successfully addressed.

This also means that the grammar is written down only once even when several aspects are applied (see the previous section). But if we look at the aspects, we see that the patterns carry on some extracts from the grammar thus it is not so obvious whether our approach helps against the problem of *duplication* or not. Let us examine this in more details using the examples from the previous section.

From the perspective of grammar duplication, the worst case is an aspect where all the patterns are exact citations from the grammar (no wildcards are used, see Listing 1.8). This means that a large part of the grammar is completely duplicated by those patterns. But if we compare this with the case of conventional annotated grammars, there still is at least one advantage of using Grammatic. Consider the scenario when the grammar has to be changed. In case of conventional annotated grammars, the same changes must be performed once for each instance of the grammar and there is a risk of inconsistent changes which are not reported to the user. In Grammatic, on the other hand, a developer can control this using *multiplicities*: for example, check if the patterns do not match anything in the grammar and report it (since the default multiplicities require each pattern to match at least once, this will be done automatically). Thus, even in the worst case, grammatical aspects make development less error-prone.

Using wildcards and subpatterns as it is done in Listing 1.7 (i) reduces the duplication and (ii) makes a good chance that the patterns will not need to be changed when the grammar changes. For example, consider the first annotation rule from Listing 1.7: this rule works properly for both Java version 1.4 and version 5 (see Listing 1.9 and Listing 1.6 respectively). The pointcut used in this rule is sustainable against renaming the symbol on the left-hand


```
classDeclaration
    : 'class' IDENTIFIER ('extends' type)?
                         ('implements' typeList)? classBody ;
```

Listing 1.9. Class declaration rule in Java 1.4

side (`classDeclaration` was renamed to `normalClassDeclaration`) and structural changes to the right-hand side (type parameters were introduced in Java 5). The only requirement is that the definition should start with the `'class'` keyword followed by the `IDENTIFIER`.

In AOP, the duplication of effort needed to modify pointcuts when the main program changes is referred to as the *fragile pointcut problem* [14]. Wildcards and subpatterns make pointcuts more *abstract*, in other words, they widen the range of join points matched by the pointcuts. From this point of view, wildcards help to abstract over the contents of the rule, and subpatterns — over the positions of particular elements within the rule. The more abstract a pointcut is, the less duplication it presents and the less fragile it is.

The most abstract pointcut does not introduce any duplication and is not fragile at all. Unfortunately, it is also of no use, since it matches any possible join point. This means that eliminating the duplication completely from patterns is not technically possible. Fortunately, we do not want this: if no information about a grammar is present in an aspect, this makes it much less readable because the reader has no clue about how the annotations are connected to the grammar. Thus, there is a trade-off between the readability and duplication in grammatical aspects and a developer should keep pointcuts as abstract as it is possible without damaging readability.

To summarize, our approach allows one to keep a context-free grammar completely clean by moving annotations to aspects and to avoid any unnecessary duplication by using abstract pointcuts.

6 Related work

Several attribute grammar (AG) systems, namely JastAdd [5], Silver [15] and LISA [13], successfully use aspects to attach attribute evaluation productions to context-free grammar rules. AGs are a generic language for specifying computations on ASTs. They are well-suited for tasks such as specifying translators in general, which require a lot of expressive power. But the existence of more problem-oriented tools such as Pretzel [3] suggests that the generic formalism of AGs may not be the perfect tool for problems like generating pretty-printers. In fact, to specify a pretty-printer with AGs one has to produce a lot of boilerplate code for converting an AST into a string in concrete syntax. As we have shown in Section 4, Grammatic facilitates creation of such problem-oriented tools providing the syntactical means (grammatical aspects) to avoid tangled grammars and unnecessary duplication.

The MPS [6] project (which lies outside the domain of textual languages since the editors in MPS work directly on ASTs) implements the approach which is very close to ours. It uses aspects attached to the *concept language* (which describes abstract syntax of MPS languages) to provide input data to generators. The implementation of aspects in MPS is very different from the one in GRAMMATIC: it does not use pointcuts and performs all the checking while the aspects are created.

There is another approach to the problems we address: parser generators such as SABLECC [2] and ANTLR [12] can work on annotation-free grammars and produce parsers that build ASTs automatically. In this way the problems induced by using annotations are avoided. The disadvantage of this approach is that the ASTs must be processed manually in a general-purpose programming language, which makes the development process less formal and thus more error-prone.

7 Conclusion

Annotated grammars are widely used to specify inputs for various generators which produce language support tools. In this paper we have addressed the problems of tangling and duplication in annotated grammars. Both problems affect maintainability of the grammars: tangled grammars take more effort to understand, and duplication, besides the need to make every change twice as the language evolves, may lead to inconsistent changes in different copies of the same grammar.

We have introduced *grammatical aspects* and showed how they may be used to cope with these problems by separating context-free grammars from annotations.

The primary contribution of this paper is a tool named GRAMMATIC which implements an aspect-oriented approach to specification of annotated grammars. GRAMMATIC provides languages for specifying grammatical pointcuts, advice and aspects.

We have demonstrated how GRAMMATIC may be used to generate a syntax highlighter and a pretty-printer by applying two different aspects to the same grammar. We have shown that grammatical aspects help to "untangle" grammars from annotations, and eliminate the unnecessary duplication. The possible negative impact of remaining duplication (necessary to keep the aspects readable) can be addressed in two ways: (i) *abstract patterns* reduce the amount of changes in aspects per change in the grammar, and (ii) *multiplicities* help to detect inconsistencies at generation time.

One possible way to continue this work is to support grammar adaptation techniques [10] in GRAMMATIC to facilitate rephrasing of syntax definitions (e.g., left factoring or encoding priorities of binary operations) to satisfy requirements of particular parsing algorithms.

Another possible direction is to generalize the presented approach to support not only grammars, but also other types of declarative languages used as inputs for generators, such as UML or XSD.

References

1. The Eclipse Foundation. Xtext. `http://www.eclipse.org/Xtext`, 2009.
2. Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society.
3. Felix Gärtner. The PretzelBook. Available online at `http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/` (last accessed on April 28, 2010), 1998.
4. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.
5. Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
6. JetBrains. Meta Programming System (MPS). `http://www.jetbrains.com/mps`, 2009.
7. Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1979.
8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
9. Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
10. Ralf Lämmel. Grammar adaptation. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 2001.
11. Leon Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.
12. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
13. Damijan Rebernak and Marjan Mernik. A tool for compiler construction based on aspect-oriented specifications. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 11–16, Washington, DC, USA, 2007. IEEE Computer Society.
14. Maximilian Störzer and Christian Koppen. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.
15. Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *ENTCS*, 203(2):103–116, 2008.

Information Systems Integration