Domain-Specific Languages for Embedded Systems Portable Software Development

Vera Ivanova, Boris Sedov, Yuriy Sheynin, Alexey Syschikov
Saint Petersburg State University of Aerospace Instrumentation
Saint Petersburg, Russia
{vera.ivanova, boris.sedov, sheynin, alexey.syschikov}@guap.ru

***Abstract*—In this paper we present a new method of Domain Specific Language development for a portable software development for embedded systems. Domain Specific Languages allow to involve domain experts apart with programmers in embedded software development. We propose a visual programming approach and methods for coarse-grained programming. In a combination with a method for domain specific languages development it allows rapid building of an infrastructure for selected domains programming. Coarse-grained approach provides abilities for easy targeting of developed program to various target platforms and configurations.**

I. INTRODUCTION

Modern fast-growing market of embedded systems offers a large number of embedded hardware solutions. Writing code personally for each embedded solution is very tricky.

Embedded systems can cope with rather wide area of problems, and the number of these areas only grows. Description of the solution issue becomes more and more important.

Implementation of some programming algorithm can be done in two ways – by using general approach and general-purpose tools or by using specialized tools and instruments.

General approaches and tools are oriented on a wide range of problems (often – for a class of problems). Usually they are not optimized for each particular problem, especially when the problem is non-trivial and has a lot of individual aspects. Specialized tools and instruments are oriented on a specific area and provide solutions only for this area. They are optimal and effective for solving this kind of problems.

Domain-specific language (DSL) is a programming or modeling language designed for a particular domain area. Unlike general-purpose languages, DSLs are more expressive, easier to use and more understandable to the different user categories. DSL allows operating with terms of their domain.

There are a lot of approaches to create new programming languages, but they are more suited for general-purpose languages. This article reviews approaches and methods of domain-specific language design and proposes a new method for DSL development.

A number of proposed hardware for embedded systems market and necessity to take maximum characteristics from this hardware generates highly specialized professionals, which have deep understanding of development process for a particular hardware platform. As a result the specialization becomes narrower. To create an effective solution for embedded system we need two specialists, first – domain specialist, second – programming specialist, who could create effective realization of domain specialist's solution on a hardware platform. One should take into account that prospective embedded platforms are multicore processors and manycore heterogeneous SoCs. Thus a DSL should support parallel programming for such platforms, with a selection of application algorithms, adaptation and tuning in respect to the platform features and characteristics.

There is a need in a technology and design tools of portable software development for embedded systems, an environment, where domain expert could describe solution for a task, and program specialist, who could implement this solution on a particular hardware platform. For this the DSL development method with minimal efforts is needed.

II. STATE OF ART

The continued miniaturization of computing devices has contributed to making embedded systems a wide variety of diverse computational requirements [1]. Regardless of the device, be it mobile phones, vehicle equipment, medical instruments, or smart home components, all of these systems embody very stringent requirements in terms of reliability, maintainability, availability, safety, security, efficiency, energy consumption, among others. Overall, the diversity of embedded systems and requirements pose tremendous challenges to the development and robustness of their software applications. In particular, this software must operate within acceptable performance parameters in resource-constrained environments while being subject to changing operating conditions (e.g., temporary unavailability of sensors, decreasing battery level, real-time

ISSN 2305-7254

requirements, memory limitations, intermittent connectivity).

Development of embedded systems in research and industry is more and more shifting from code based development to model driven development [2] (MDD) approaches, which are founded on high-level modeling languages. Modeling languages are not as generic as general purpose programming languages, they provide more specialized language constructs, e.g. for the creation of data flow based systems (e.g. Simulink [3]) or for the creation of system models (e.g. SysML [4]). These MDD approaches are supported by industrial strength tool chains; prominent examples of MDD tools that are applied in both research and industry are Simulink, ASCET [5], SCADE[6], Rhapsody[7], Artisan[8], and MagicDraw [9]. MDD tools implement modeling languages, provide infrastructure support, e.g. tailored editors and code generators, and include runtime libraries and frameworks that support execution of generated code.

There is a major challenge for the development of embedded systems: generic and domain specific modeling languages are limited and support some aspects of embedded system development only. Simulink, for example, supports definition of data flow based behavior only, UML [10] based languages support the definition of software architecture and control flow, and SysML supports the definition of system architectures. Graphical editors, code generators, and language frameworks only support one or a limited set of modeling languages. Detailed modeling of all aspects of complex embedded systems therefore requires the combination of models defined in multiple modeling languages and tool chains to provide one holistic system model. Code generation needs to be done with multiple independent generators in this case. This yields the situation that developers need to combine multiple generated artifacts and runtime libraries, and need to connect required inputs and provided outputs of models, which may even implement different semantics. One common execution model is required that supports all relevant modeling languages. This non-trivial task currently limits the applicability of DSL approaches in development of complex software systems, since the effort required for integrating modeling languages may outweigh the additional benefits of modeling languages

III. Overview of DSL development methods

DSL development consists of following phases: decision about necessity of a new DSL, domain analysis, language design, implementation and deployment [11]. In fact, the development of a domain-specific language is not a simple sequential process. Decision phase can be influenced by preliminary analysis, which can also weigh with design part, and design can be affected by implementation considerations. Each phase is associated with a set of patterns, except for deployment phase, which is left behind the scope of this article.

Development process also can be separated in two parts: “when should you develop DSL” and “how should you develop DSL” (Fig. 1).

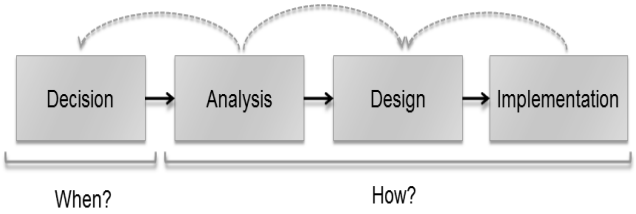

Fig. 1. DSL development phases

Decision phase corresponds to the “when” part, other phases correspond to the “how” part.

Decision patterns represent the set of situations when it’s rational to use DSL. It is obvious, that application of existing DSL is less expensive and requires less experience, than developing a new one.

In the analysis phase of DSL development the problem domain is identified and domain knowledge is collected. Sources are technical documents, domain experts’ knowledge, legacy code in GPL, user feedback and so on.

DSL design approaches can be considered from two points of view: the relationship between the DSL and existing languages, and the formal nature of the design description.

From the first point of view there are two ways to design DSL – inventing a whole new language or using some existing language as a base. Inventing a new language can be complicated and expensive. That’s why preference is usually given to the second approach. Second approach, in turn, is divided into three patterns [12].

1) Piggyback. Base language constructions are supplemented by new DSL constructions, and then it’s compiled (translated) into host language code (not into native code). In the same time host constructions stay unchanged, but DSL instructions are transformed to base language instruction. Typical examples of this approach are the yacc [13] and lex [14] processors. While the specifications of the input grammar (in the case of yacc) and the input strings (in the case of lex) are expressed in a DSL, the resulting actions for recognized grammar rules and tokens are specified in C which is also the processors’ output language. Yacc uses the piggyback approach more aggressively as it introduces special variables (denoted by the $ sign) to the C constructs used for specifying the actions.

2) Language Specialization. Some instructions which impede an adaptation to the particular domain are removed from host language. In some cases the full power of an existing language may prevent its adoption for a specialized purpose. A representative case arises when requirements related to the safety or security aspects of a system can be satisfied only be removing some “unsafe” aspects (such as dynamic memory allocation, unbounded pointers, or threads) from a language. In such cases a DSL may be designed and implemented as a subset of an existing language. Examples of DSLs designed following the specialization pattern are Javalight [15] which is a type-safe subset of Java, the educational subsets of Pascal used for a stepwise introduction to the language [16], the HTML application of SGML [17], and the automotive “safer-subset” of C [18].

3) Language Extension. This pattern is used to add new features to an existing language. Often an existing language can effectively serve a new need with the addition of a few new features to its core functionality. In this case, a DSL can be designed and implemented as an extension of the base language. The language extension pattern differs from the piggyback pattern by the roles played by the two languages: the piggyback pattern uses an existing language as an implementation vehicle for a DSL, whereas the extension pattern is used when an existing language is extended within its syntactic and semantic framework to form a DSL. One of the earliest examples of this pattern is the “rational FORTRAN"' (Ratfor) compiler [19] which provided a structured version of FORTRAN. The implementation of the original C++ compiler (cfront) [20] also used this technique.

From the second point of view DSL designer must choose how to specify the design before implementation - in formal or informal way. In an informal design the specification is usually in some form of natural language, probably including a set of illustrative DSL programs. A formal design consists of a specification written using one of the available semantic definition methods [21]. The most widely used formal notation include regular expressions and grammars for syntax specifications, and attribute grammars, rewrite systems and abstract state machines for semantic specification.

Clearly, an informal approach is likely to be easiest for most people. However, a formal approach should not be discounted. Development of a formal description of both syntax and semantics can bring problems to light before the DSL is actually implemented. Furthermore, formal designs can be implemented automatically by language development systems and tools, thereby significantly reducing the implementation effort.

After completing DSL design one should choose suitable implementation. It is difficult due to common approaches to GPL implementation are not applicable to DSLs. Patterns for DSL are not widely known, but some of them are presented below [22].

4) Interpreter. DSL constructs are recognized and interpreted by standard “fetch-decode-executed” cycle. This approach is convenient for languages having a dynamic character or if executing speed is not important. The advantages of interpretation over compilation are easier extension and greater simplicity.

5) Compiler/application generator. DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/specification. DSL compilers are often called application generators.

6) Preprocessor. DSL constructs are translated to constructs of an existing language. Static analysis restricted to abilities of base language processor. Important sub-patterns:

- Macro-processing: extension of macro-definitions into plain code.
- Source-to-source transformation: The DSL source code is transformed via a suitable shallow or deep translation process into the source code of an existing language. The tools available for the existing language are then used to host - compile or interpret - the code generated by the transformation process.
- Pipeline: In cases where a number of DSLs are needed to express the intended operations, their composition can be designed and implemented using a pipeline. Typically, all DSLs are organized as a series of communicating elements. Each DSL handles its own language elements and passes the rest down to the others. Sometimes, the output of one DSL can be expressed in terms of the input expected by another DSL further down the pipeline chain.
- Lexical processing: Many DSLs can be designed in a form suitable for processing by techniques of simple lexical substitution; without tree-based syntax analysis. The design of the DSL is geared towards lexical translation by utilizing a notation based on lexical hints such as the specification of language elements (e.g. variables) using special prefix or suffix characters. The form of input for this family of DSLs is often line-oriented, rather than free form and delimited by character tokens.

7) Embedding. DSL is implemented by extending an existing GPL (the host language) by defining specific abstract data types and operators. A domain-specific problem can then be de-scribed with these new constructs. Therefore, the new language has all the power of the host language, but an application engineer can become a programmer without learning too much of it. To approximate domain-specific notations as closely as

possible, the embedding approach can use any features for user-definable operator syntax the host language has to offer. Application libraries are the basic form of embedding.

8) Extensible compiler/interpreter. A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.

IV. The proposed DSL development method

A. Description

After we had analyzed all described above methods, we developed our own. Remembering that development of DSL from scratch is more complicated, than development of DSL, which is based on existing language; we decided to take general-purpose visual language, which could be used by domain experts to create their own new DSLs with minimal efforts, as a base. In the implementation of these languages programmers take place by using applications generation approach (compiler/application generator).

B. Advantages of a visual approach

Visual representation of computation load requires adequate approaches, considering a multitasking nature of a typical set of computational (informational, communicational, interface) tasks of a particular domain. The analysis of successful existing approaches proves the expediency of using visual programming for designing a high-level algorithmic description of a computation load as a system of communicating processes. It makes native algorithms and tasks parallelism representation easy and natural.

Despite of constant advancing of high-level languages, programming usability issues and work speed issues remain rather relevant today. It can explain popularity and a wide distribution of graphic visual programming. It provides maximal level of abstraction allowing users to work more efficiently, especially when multitasking software package is designed in terms of algorithms and flow-charts.

C. Intuitive use of a graphic

As most people, engineers and scientists solve their tasks by operating with images and symbols of a problem domain. Such an approach has been developed in the process of education and application of the relevant data processing tools, such as charts and diagrams. However, most programming languages require the study of specific syntax and adaptation of a domain model to language features. At the same time graphic language allows to work with the intuitive structures.

Graphic language code is usually more suitable for engineers and scientists, because they usually work with a visual data; process modeling by flow-charts and state diagrams which also show the data flows. Besides, stream programming calls for a work in terms of problem domain. For example, a typical application written in graphic language firstly receives data from the several temperature sensors, then passes this data to analysis function, and then, finally, stores the computed data (Fig. 2.) Graphic representation of this program is clearly defines execution order of operations and data flows.

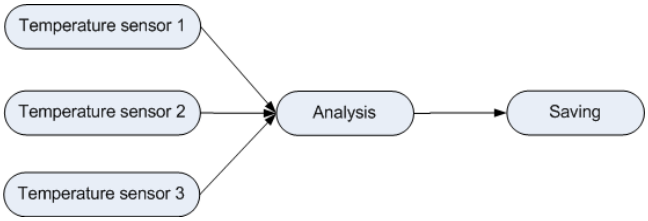

Fig 2. Typical application written in graphic language

Graphic language conception is easy to understand, so the tools of development environment can be made just as suitable and intuitive. For example, debugging tools can visualize the process of data distribution through the channels, and also display the appropriate values on inputs and outputs of program code nodes (what is meant here is an interactive background of the execution).

Debugging tools allow setting breakpoints in many parts of program simultaneously, pausing the execution and entering the procedure. Many development environments for textual languages have a similar functional, but graphic language environment can display current state of the program and relations between parallel code sections in a more convenient form due to graphic core of this language.

D. Natural parallelism

Unlike sequential execution languages such as C and C++, graphic languages initially contain information about code sections which can and cannot be executed in parallel (Fig. 3). It allows exploiting the architectural advantages and simplifies the task of flow creation and control for domain specialist.

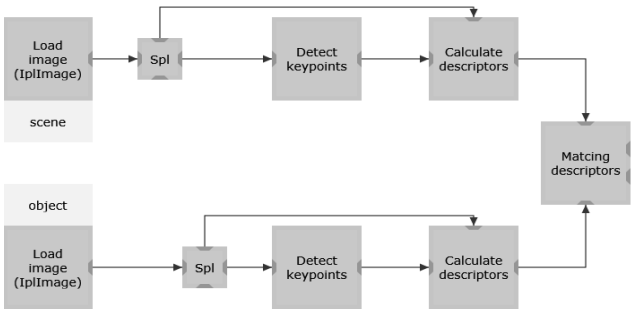

Fig 3. Natural parallelism on visual language

A distinct advantage of graphic languages compared to the usual textual languages is that multithreaded applications implementation becomes a simpler task. Graphic language

compiler can independently determine code sections having parallel blocks and organize separate threads for their parallel execution. In computer terminology such mechanism is called an "implicit parallelism", i.e. the parallel implementation is performed by development tools on the basis of program approach's features, not by a specially written parallel program code.

E. Automated pipelining

Due to own structure, graphic language code could be naturally pipelined during the runtime. Understanding of this fact allows the developer to identify potential stages of the pipeline. The process of distributing the program among blocks (future pipeline stages) is clear and obvious due to block structure of a graphic language (Fig. 4).

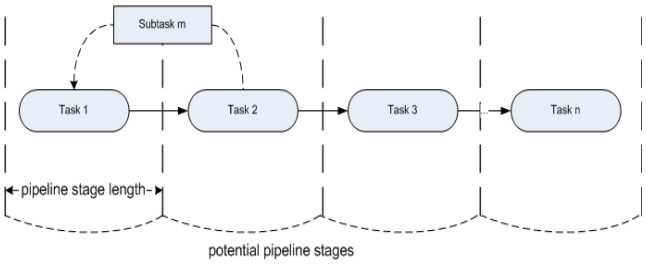

Fig 4. Automated pipelining on visual language

The developer is able to influence the stage length and distribution among pipeline stages by changing the granularity level of a visual code or by redistribution of blocks, which eventually allows making maximum use of target platform features.

F. An integration of graphic and textual languages

Despite the fact that graphic languages are well-suited for the organization of a code parallelization, and they hide the nuances of memory management, they are not always suitable for solving some problems. In particular, mathematic formulas and equations can be represented more vividly in a textual form, that's why graphic language unit can have several representation levels: external and internal (Fig. 5).

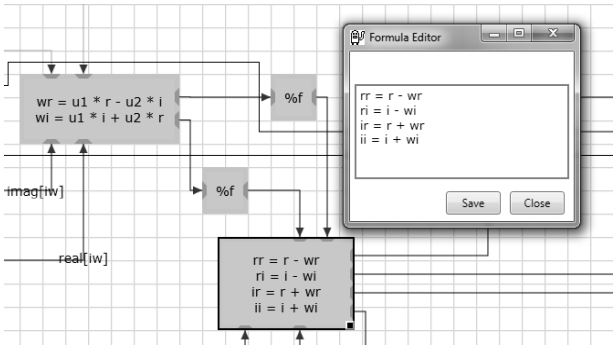

Fig 5. Representation of graphic language unit

External representation points out that this element describes, for example, mathematic formula; internal representation captures the essence of unit – a textual formula.

G. Granularity

Graphic language concept implies the following scheme of program development. Design starts with a coarse-grained scheme. Problem originator collects all the coarse stages needed for the problem solution into a single scheme. Then every block is detailed. Problem originator achieves such level of granularity when every block gets a small amount of an input data and processes them in a relatively simple manner. Also target platform significantly affects the choice of granularity level. This approach allows making every block autonomous in a sense that its implementation doesn't need to take into account all relations in the program. One only needs to implement a restricted number of operations under a particular input data. Due to such localization, problem originator can pass the blocks to programmer, who will implement them with a visual parallel language, textual language, or with their combination. After blocks implementation the developer gets a working program.

Such approach has many advantages. Firstly, the program becomes for suitable for further changing and maintenance. Due to block structure, the developer has an opportunity to rearrange blocks without losing their working capacity. It considerably increases design flexibility. Secondly, the smaller is granularity, the greater is possibility of a potential program parallelization during the runtime.

H. Our method of DSL development

We propose VPL (Visual Programming Language) [23,24] as a base for development of new domain-specific languages. This language is based upon AGP-model (a model of Asynchronous Growing Processes) [25,26].

Basic elements of VPL have rather low level of abstraction, but one can describe coarse-grained and medium-grained elements by means of hierarchical program structuring tools built in the language (Fig. 6). This is how the procedures of particular domain are transformed into functional blocks making up final program.

A set of functional blocks representing the most frequently used domain functions and a set of basic elements will make up a new DSL by using language extension pattern. The elements of the base language can be either hidden or visible, depending on the domain characteristics.

A realization of a new domain-specific language requires no additional efforts since the new semantic elements can be produced in two ways.

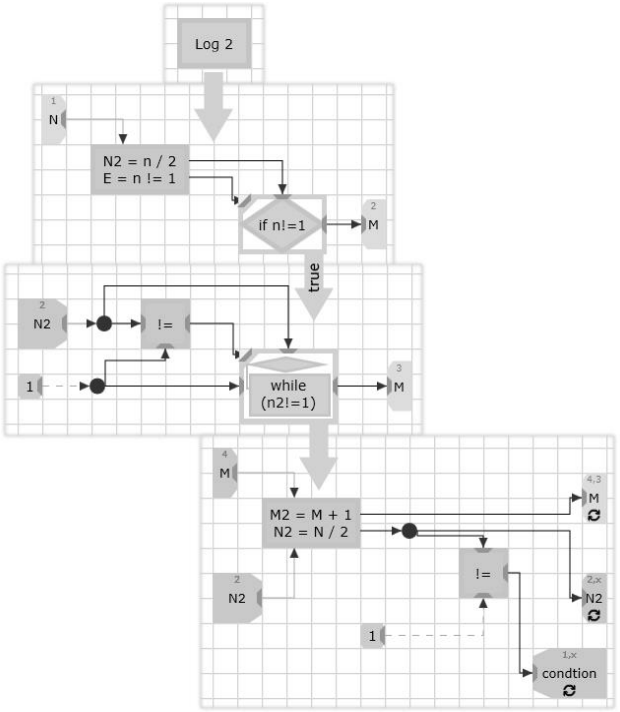

Fig 6. Low level description of library node

Firstly, elements are composed of basic language units and transformed from VPL to an internal representation by general rules. Then the internal representation will go to the input of different compilers, which produce the target platform code. Thus, domain-specific languages that were constructed on the base of VPL will be implemented by means of application generators.

Second, elements refer to some function written in general-purpose language. So this function is assigned to a particular element by special rules. This reference is saved during transformation to the internal representation. Function code will be inserted instead of generated code during internal representation transformation to target platform code.

In this case one of the main requirements to DSL is observed: domain experts don't have to care about functional blocks implementation. This implementation is created by programmers in the target platform language. There can be several implementations and switching between them doesn't require much effort both from expert and programmer.

V. Conclusion

The proposed method is based on using visual programing language VPL as a base for developing of new domain-specific languages. In this method new DSLs are developed by using language extension. Thus DSLs allow domain specialist and programmer work simultaneously and independently on the same scheme.

Acknowledgment

The research leading to these results has received funding from the Ministry of Education and Science of the Russian Federation under agreement n°14.575.21.0021.

References

[1] T. Kuhn et al., "Multi-Language Development of Embedded Systems", in *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM '09)*, M. Rossi et al., Eds., pp. 21-27.

[2] G. Trombetti, A. Gokhale, D. C. Schmidt, J. Hatcliff, G. Singh, J. Greenwald "A Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems," in *Model Driven Software Development- Volume II of Research and Practice in Software Engineering* , S. Beydeda, M. Book, and V. Gruhn, Eds. New York: Springer-Verlag, 2005.

[3] Pieter J. Mosterman. *MATLAB and Simulink for Embedded Systems Design.* The MathWorks, Inc. 2007

[4] Rosenberg D*., Mancarella S. Embedded system development using SysML.* OMG: Systems Modeling Language 2010.

[5] ETAS - ASCET Software Products - http://www.etas.com/en/products/ascet_software_products.php

[6] Esterel Technologies SCADE Suite. The Standard for the Development of Safety-Critical Embedded Software in Aerospace & Defense, Rail Transportation, Energy and Heavy Equipment Industries. // Esterel Technologies - Critical Systems and Software Development Solutions. — http://www.esterel-technologies.com/products/scade-suite.

[7] Rational Rhapsody Architect for Software - http://www-03.ibm.com/software/products/ru/ratirhaparchforsoft

[8] Artisan Software Tools Inc. http://www.atego.com/products/technology-overview/

[9] MagicDraw - No Magic, Inc http://www.nomagic.com/products/magicdraw.html

[10] Chen R., Sgroi M., Lavagno L., Martin G., Sangiovanni-Vincentelli A., Rabaey J. Embedded System Design Using UML and Platforms. In *Forum on Specification & Design Languages, Marseille*, France, September 2002.

[11] Marjan Mernik, Jan Heering, Anthony M. Sloane: When and how to develop domain-specific languages. *ACM Comput. Surv*. 37(4): 316-344 (2005).

[12] Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001

[13] Stephen C. Johnson. Yacc - yet another compiler-compiler. *Computer Science Technical Report 32*, Bell Laboratories, Murray Hill, NJ, USA, July 1975.

[14] Michael E. Lesk. Lex - a lexical analyzer generator. *Computer Science Technical Report 39*, Bell Laboratories, Murray Hill, NJ, USA, October 1975.

[15] Tobias Nipkow and David von Oheimb. Javalight is type-safe-definitely. In Conference Record of POPL '98: *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161-170, San Diego, California, 1998.

[16] Walter Savitch. Pascal - An Introduction to the Art and Science of Programming. Benjamin/Cummings Pub. Co., Inc., fourth edition, 1995.

[17] International Organization for Standardization, Geneva, Switzerland. *Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*, 1986. ISO 8879:1986.

[18] P. D. Edwards and R. S. Rivett. Towards an automotive `safer subset' of C. In Peter Daniel, editor, *16th International Conference on Computer Safety*, Reliability and Security: SAFECOMP '97, pages 185-195, York, UK, September 1997. European Workshop on Industrial Computer Systems: TC-7, Springer Verlag.

[19] Brian W. Kernighan. *Ratfor - a preprocessor for a rational Fortran.* Software: Pract. & Exp., 5(4):395-406, 1975.

[20] James M. Stichnoth and Thomas Gross. *Code composition as an implementation language for compilers.* P.119-132.

[21] Slonneger,K. Kurtz, B. L. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley. 1995.

[22] Kamin, S. 1998. Research on domain-specific embedded languages and program generators. Electro. Notes Theor. Comput. Sci. 14 . http://www.sciencedirect.com/.

[23] Sheynin Yuriy E., Syschikov Alexey Y. *Enabling graphical notation for parallel programming : 2009/0064115 A1.* — United States, March 5, 2009

[24] Sheynin Yuriy E., Syschikov Alexey Y. Parallel programming language for coarse-grained dynamic computations // *In Proceeding of the 3rd International Conference "Parallel computations and control tasks"* – Moscow, 2006.

[25] Sheynin Yuriy E. Asynchronious Growing Processes – a formal model of a parallel computations in distributed computing structures *// Distrubuted data processing. In Proceedings of the International Conference ROI-98* – Novosibirsk, 1998 – p. 111-115.

[26] Sheynin Yuriy E. *A formal model of dynamic parallel computations in parallel computation systems for experimental data processing* // Scientific Instrumentation – 1999 – v. 9, 2 – p. 22-29.