Tiny Programming Language to Improve Assembly Generation for Automation Equipments

José Metrôlho, Mónica Costa, Fernando Reinaldo Ribeiro

***Abstract*—** The development time in industrial informatics systems, within industry environments, is a very important issue for competitiveness. The usage of adequate target-specific programming languages is very important because they can facilitate and improve developers' productivity, allowing solutions to be expressed in the idiom and at the level of abstraction of the problem's domain.

In this paper we present a target-specific programming language, which was designed to improve the design cycle of code generation, for an industrial embedded system. The native assembly code, the new language structure and their constructs, are presented in the paper. The proposed target-specific language is expressed using words and terms that are related to the target's domain and consequently it is now easier to program, understand and to validate the desired code. It is also demonstrated the language efficiency by comparing some code described using the new language against the previous used code. The design cycle is improved with the usage of the target-specific language because both description and debug time are significantly reduced with this new software tool. This is also a case of university-industry partnership.

***Keywords*—**Compilers and Interpreters, Embedded Systems, Industrial Systems, Programming Languages, Software Design and Development.

I. Introduction

THE number of companies producing software has grown constantly and the need of software increases more and more every day. The importance of cost efficiency relationship and creation value in software development as well as in software process and product improvement is a central feature; companies have noticed that competition is increasing and cost-efficiency companies have perhaps more competitive advantage in global markets than ever [1]. The development time in industrial informatics systems, in industry environments, is a very important issue for competitiveness. Companies that develop solutions for industry usually deal with several levels of abstractions, from high level languages to assembly. As we move towards the high to low level languages the effort is greater and the developers generally want to work with more abstract levels. However, it is very common for these companies to handle with specific embedded devices, that require specific programming languages, mainly low level programming languages. Although low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the microprocessor/microcontroller, increasing its efficiency, writing a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. It requires a deep understanding of the microprocessor concepts to produce reliable and maintainable programs. These skills can only be expected from professional software developers.

As is corroborated by Preuer [2], restricting the focus to a specific problem domain allows the application of domain-specific concepts and techniques that enable domain experts to develop software without being professional programmers. In this scenario Domain-Specific Languages (DSL) and Target-Specific Languages (TSL) can play an important role in facilitating the software developers' task increasing their productivity. DSL and TSL are programming languages for solving problems in a particular domain. They are much more expressive in their domain and allow faster development of programs allowing solutions to be expressed in the idiom and at the level of abstraction of the problem's domain. DSL and TSL provide several advantages over general purpose programming languages, namely [3] concrete expression of domain knowledge, direct involvement of the domain expert, expressiveness, modest implementation cost, reliability, training costs and design experience. These types of programming languages are usually small, more declarative than imperative and more attractive than general-purpose languages because of easier programming, systematic reuse, better productivity, reliability, maintainability and flexibility. DSL and TSL bring programming closer to application domains and have the capability to significantly improve the productivity and quality of software engineering in the focused domain.

In this paper we describe a TSL to improve developer's productivity in industrial embedded systems in the scope of University-Industry collaboration. Preliminary tests show that the TSL decreases the development time and increases developers' productivity.

The remainder of this paper is structured as follows: in Section 2, we introduce the target environment and in Section 3 we describe the native language of the hardware. In Section 4 we present the formalism of the TSL and in Section 5 we present preliminary tests. Finally, Section 6 concludes this paper with a discussion of the pre and pos systems implementation and pointed out some directions of future work.

II. Related Work

Research on domain and target specific programming languages, for industrial informatics systems in industry environments, has received considerable attention with many projects addressing the issues such as: how to facilitate and improve the developers' productivity and how to improve the cost efficiency and value creation in software development.
Ojala [1, 4] discusses the concepts, principles and practical methods of economic-driven software engineering and outlines us to understand better the content of value-based approach. This is done in part by presenting a conceptual analysis of the economic-driven view of software development, including cost estimation and cost accounting, and in part by discussing the cost efficiency and value characteristics of software processes, products and their improvement.

Babcicky [5] developed a special purpose programming language which is an object-oriented language, semantically heavily inspired by SIMULA, which became known as TESLA (TEst Scripting LAnguage). It has an ALGOL originated block structure with sub-blocks, procedures and classes with inheritance, dot notation for accessing object attributes and methods and the customary set of statement types, including the connection statement. The author also lists some benefits and drawbacks of the effort related to the development of a new language and points out some important aspects that should be taken into account, namely the difficulty of achieve a final solution at the first attempt and also the language and system promotion in a way to gain programmers support and acceptance.

Prähofer et al. [6, 7] present the language Monaco, which is a domain-specific language for programming reactive control programs. The main purpose of the language is to bring automation programming closer to the domain experts and end users. Important design goals therefore have been to keep the language simple and allow writing programs which are close to the perception of domain experts.

In [8] F. Wenzel and R.-R. Grigat introduce a framework for developing image processing algorithms. Its design is targeted at the needs of developers who should be able to focus on their specific tasks as much as possible instead of technical side effects that arise in software development. They point out two aspects of their approach. First developers are not required to gain knowledge of foreign domains like GUI programming. Secondly the source code for new methods can be kept in a future-proof way.

Although there exist several works and projects that studied and proposed new programming languages for specific domains, each one presents their own particularities because they want to be well fitted to a particular environment with specific users, interests and specific equipments. In this work the proposed TSL is targeted to a company that develops industrial informatics solutions for other companies, mainly to the automotive industry. The company presents its own organizational culture and uses specific equipments, and one important design goal of the new language is to facilitate and improve developers' productivity allowing solutions to be expressed in the idiom and at the level of abstraction of the problem's domain.

III. Why do we need a target-specific language programming

The challenge proposed by the target company is related to the improvement of developers' productivity with respect to the process of programming their hardware modules which are currently programmed through a low level language that is very time consuming and require a deep understanding of their concepts.

The possibility of develop a new programming language was carefully analyzed and the main question that needed to be clarified was: is it worthwhile to develop a specific programming language?

Clearly there were the general benefits of using target specific programming languages, that are presented in previous section (e.g. expressiveness, modest implementation cost, reliability, training costs and design experience), but some other aspects were taking into consideration before deciding develop a new and specific programming language. Develop a new programming language poses some risks related to: the complexity of language design and implementation; learning effort associated with using a new programming language and high startup costs (due to the complexity of design and implementation) notwithstanding the fact that usually allows applications to be developed more cheaply afterwards [9] (see figure 1).

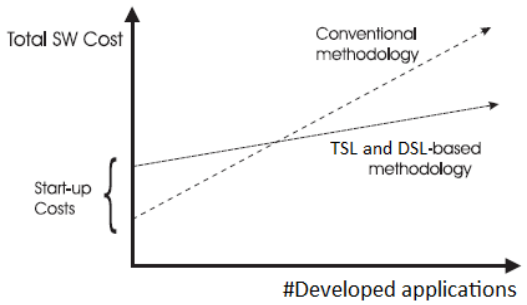

Fig. 1 The payoff of DSL methodology (adapted from [10]).

On the other hand, creating a target-specific language can be worthwhile if the language allows a particular type of problems or solutions to them to be expressed more clearly than pre-existing languages. This is the case of this situation because: the introduction of a new target specific language, focused to a specific problem domain, allows programmers to develop their applications faster and thus increasing their productivity; by having a syntax that is understandable to non-programmers, it may allow domain experts to program applications themselves [9].

IV. THE TARGET ENVIRONMENT

Due to confidential constraints we will not present details about the module used by the company. This company develops industrial informatics solutions for other companies, mainly to the automotive industry. But in general terms, and to introduce the theme, we can inform that the target module (see figures 2 and 3) is used to actuate over relays and has several internal units like timers and I/O ports that can be configured using a dedicated assembly language. Some module features are: 6 Digital I/O pins; 3 Transistor Outputs; 1 Relay outputs; 2 Analog inputs; 1 counter and 8 32 bit timer with a time resolution of 1 ms.

The hardware module has characteristics of a modular system and multiple modules can be connected in bus topology. This characteristic makes it ideal for wiring tests either in the prototype stage (cable design, allowing adjustments in the location of components) either in the production phase (test various options of a cable, like left or right steering-wheel), because they offer flexibility to the test table in terms of layout. Another functionality of the module is the component mechanical/electromechanical reliability test. The module can be used to many operations like to enable/disable outputs, as well as the reading of digital inputs and perform different wiring tests, for example the number of cycles that a relay is flawless. The main functionalities of the module are: continuity tests; measurement of resistance/capacity; activation of outputs and reading inputs/outputs.

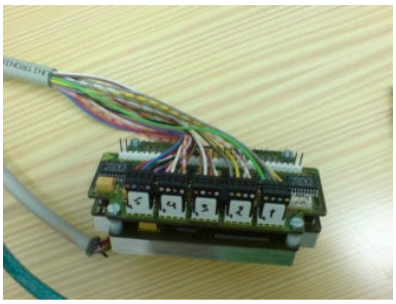

Fig. 2 Hardware module.

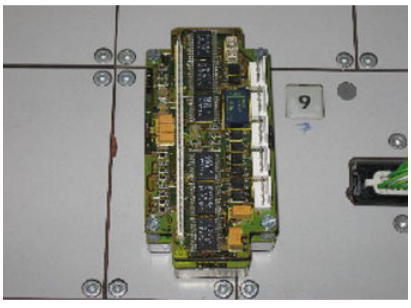

Fig. 3 Hardware module installed in the test table.

Those modules have a set of registers whose bits have particular meanings. These registers can be of different types: read, write or read/write. A feature of the assembly language is that any time the designer wants to read or write something, he must knew the register number and each the bits meaningful. This demands a lot of manual readings and becomes repetitive for some applications.

Another feature is that the necessary instructions to build applications are scarce and all well defined. As example a read or write relay operation is almost the same, but requires knowing the name of the register and to know the bit number that must be set or reset to act according the desired action. Additionally the code is only readable and understandable by developers that have knowledge about that particular assembly. A language that could be more intuitive and make code more documented and understandable was desired.

This leads to the idea that a high-level programming language, more adapted to the field, can be designed with proper and intuitive constructs, like in this case relay(on), or relay(off) avoiding details and constants that are well known and thus improving developers' productivity.

The development of applications, before the new tool described in this paper, was done by writing assembly code that is uploaded to the modules by a proprietary application. This fosters a deep knowledge about the assembly and about the registers and the meaning of its bits. To develops applications with a low time to market a more abstract tool is needed, this s the goal of our approach. This paper describes a tiny language designed and implemented to allow quicker developing time and also generated assembly code documented and indented properly to foster faster detection of software bugs.

V. THE NATIVE LANGUAGE

Here we present some of the assembly language features. The following piece of code (see figure 4) shows a sample of the type of details and structure which must be introduced by the programmer.

```
$init
   ...
MOVI(T0VAL,0)
MOVI(T0MAX,1000)
MOVI(T1VAL,0)
MOVI(T1MAX,500)
...
WREG(A2,5,255)
MOVI(A13,2)

$code
RREG(A4,6)
ANDI(A10,A4,8)
SRI(A10,A10,3)
ANDI(A11,A4,16)
SRI(A11,A11,4)
ANDI(A12,A4,32)
SRI(A12,A12,5)
IFEQ(T0VAL,T0MAX)
```

```
ORI(A10,A10,2)
MOVI(T0VAL,0)
ENDIF
....
$end
```

Fig. 4 Sample of native assembly code.

As it can be observed in Fig. 4, the user must be aware of the native assembly and a constant set of variables that can be used and must deal with information about the registers and also regarding timers, he/she must convert the time unit to milliseconds. These details are prone to generate errors.

So this case-study has fostered the design of a tiny language to describe applications for an embedded device that is used in industrial environments. The main goals of the new language are, transform the design of new programs as high level as possible, use intuitive constructs, allow some verifications to avoid errors, make the code documented and automatically idented. In other terms, make the design time shorter with less design effort for the designers of applications involving that embedded microcontroller.

VI. The new language

Here we will describe the developed tool. First we will present the structure and then the constructs of the new language.

A. The new language structure

The new structure has 2 sections, one for declarations and other for code. This is similar to the target assembly, however the section delimiters are now '{' as in common languages.

Within each section the user will now avoid details and will focus on actions or constructs that are common to programmers and for designers of that kind of applications. The constructs were defined to make clear the programs, and to avoid details. The tool will then generate the proper code..

B. The new language constructs

Number After studying the possible instructions and the final result in the module, we define a set of keywords to allow an easy and intuitive definition of those instructions. As example to control a digital output the bit 0 of the module register 7 must be set/reset. In assembly this is dome using the instruction WREG(A0,7,1). As we can observe the user must put the number of the target register, a variable that transport the value that must be put over the bit (ex: since A0=0 then the bit 1 will be reset), and the number of the bit that will suffer the change (in this case is the 1st bit). However based on the "clients" feedback we notice that this output is always used for relay control. So, we defined a language construct "relay" with a single switch that makes this description easy and intuitive. Next we present in the left the new language construct usage and on the right the generated/corresponding assembly.

```
relay(on);    → WREG(A0,7,1)
relay(off);   → WREG(A1,7,1)
```

Other examples of usage of the new language constructs and the corresponding assembly:

```
var A31=2;    → MOVI(A31, 2)
attr A31=A5;  → MOV(A31, A5)
IN (0,A3);    → RREG(A3, 8)
                ANDI(A3, A3, 1)
startT(0);    → MOVI (T0VAL,0)
defT(1,1500); → MOVI (T1MAX,1500)
stopT(1);     → MOVI(T1VAL,1501)
```

Fig. 5 New language constructs.

We've defined a set of keywords for the language, in small number due to the simplicity of the assembly. The total of keywords is 28 and all of them are presented in the following table.

TABLE I
LANGUAGE CONSTRUCTS

init	IN
code	INOUT_R
end	INOUT_W
var	OUTPUTS_R
attr	OUTPUTS_W
stopT	INPUTS_R
OUT	INPUTS_W
JMP	IOCTL_R
JMPI	IOCTL_W
JMPIX	rele
if	delay
elif	startT
else	setT
testTLimit	defT

This is also interesting because a small set of keywords represents a small time to learn the language.

C. The generation chain

To implement this code converter, from the new language to the target assembly, the software chain can be represented as in Figure 6.

The code was developed using Java [11] and within the Eclipse IDE [12]. To implement the lexer and parser we used ANTLR (ANother Tool for Language Recognition) [13]. It provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages [13] including Java.

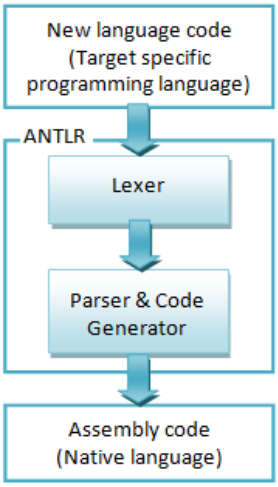

Fig. 6 Generation chain.

These choices were made to obtain an independent application platform using free software. The integration of these tools to build the previously presented generation chain was straight and software consistency was achieved.

As software development methodology Scrum [14] and XP [15] were used to achieve a short time to market application. The involved team was constituted by 4 members and the client. The client was the company representative that helps the team to reach the goals as exactly desired by the target users. The scrum's sprint time was 1 month and the application has 2 releases. One after the first 2 work months and the second in the end of the fourth month.

The 4 members of the team were a teacher and 3 students. The reached goals were the skills that students acquire in a few directions. The first one was the experience to deal with automation, compilers, language processors, programming and software integration. The second was the opportunity to deal with professional software development methodologies as Scrum and XP. This development environment fosters a better preparation of those undergraduate students and also allowed them to be involved on the development of an application useful and complex for industrial application. The course, were students were members, is Computer engineering on the Bologna format (3 years long).

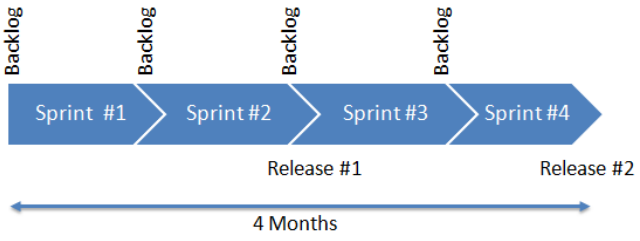

Fig. 7 Work evolution.

Due to the team members' experience, sprints were designed to be light what means that the tasks were planned with a significant time overhead. If this was made by full time developers the time to market could be significantly improved, that means that the contribution of this paper is a tiny language that can be easily adopted for similar requirements and be developed in a short period of time.

The module has a single complex problem not solved using the sequential paradigm. The delay feature is implemented as a ladder approach. However a if-then-else cascade was implemented to allow this feature. However due to some complexity of this singular problem to this application a skeleton is generated and the user must fill the generated code on the assembly to guaranty consistency of the generated program. This is the only limitation that requires operator's intervention. However, according to the client's feedback, it is a not frequently used issue in the modules, so that has considered a non priority instruction. The instruction exists and generated the skeleton in case it is necessary.

In terms of code specification and design, from the operator's point of view, the improvement was huge due to the allowed abstraction. Now the operators easily program an application using terms that are related to the module features and similarly as in the manual. This allows to focus on the desired features of the application without care about to much details as the pin order o activate or turn off an relay or other feature of the system.

VII. Tests

In terms of tests the achievement of a smaller design time was the main goal. To test it we ask the development team of the partner company to give us their feedback. The feedback was positive since the new tool allows reaching sooner and in a more proper manner the target assembly. The code becomes easily documented and the code is also readably.

In terms of the generated assembly the result is the same, as expected. However, now the user focus on the desired goals and the tool translates that for proper assembly.

In the following figures we present the code of a program in the new language and the resultant generated assembly.

```
program Exemplo{
  init{
        IN(4, A10);
        startT(7);
        startT(5);
        var A10=0;
        var A11=1;
        var A12=100;
        rele(off);
        defT(0, 5s);
        startT(0);
        defT(1,5s);
        stopT(1);
        defT(2, 4h);
        startT(2);
```

```
            defT(6, 19h);
            stopT(6);
                ...
            }
code{
          if(tstTLimit(0)){
            rele(on);
            startT(1);
            stopT(0);
          }
          if(tstTLimit(2)){
            rele(off);
            stopT(0);
            stopT(1);
            stopT(2);
            startT(4);
            var A10=1;
          }
          if(A10==A12){
            stopT(0);
            stopT(1);
            startT(3);
            var A10=0;
          }
          ...
      }
}
```

Fig. 8 New language code.

As we can notice in Figure 9, the generated code is automatically commented and formatted. This gives the designer an easier method to understand code, even at the assembly level.

Also the generated code in the native language is more extensive than the new proposed language. This means that the present effort is less than previously when descriptions were done in assembly. Also the tools detects errors and signals the line were they occur, this fosters better software development in terms of developing time and code quality. Also the reduced number of language constructs fosters a quick learning of the language descriptions features.

The generated code:

```
## Init Section
$init
# Reserved variables
   MOVI(A0,0)
   MOVI(A1,1)
# Read Input 4
   RREG(A10, 8)
   ANDI(A10, A10, 5)
# Initialize Timer T7
   MOVI(T7VAL,0)
# Initialize Timer T5
   MOVI(T5VAL,0)
# Initialize variable
   MOVI(A10,0)
# Initialize variable
   MOVI(A11,1)
# Initialize variable
   MOVI(A12,100)
# Rlay OFF
   WREG(A0,7,1)
# Timer T0 Max value
   MOVI(T0MAX,5000)
# Initialize Timer T0
   MOVI(T0VAL,0)
# Timer T1 Max value
   MOVI(T1MAX,5000)
# Stop Timer T1
   MOVI(T1VAL,5001)
# Timer T2 Max value
   MOVI(T2MAX,14400000)
# Initialize Timer T2
   MOVI(T2VAL,0)
# Timer T6 Max value
   MOVI(T6MAX,68400000)
# Stop Timer T6
   MOVI(T6VAL,68400001)
## Code Section
$code
IFEQ(T0VAL,T0MAX)
   # Relay ON
      WREG(A1,7,1)
   # Initialize Timer T1
      MOVI(T1VAL,0)
   # Stop Timer T0
      MOVI(T0VAL,5001)
ENDIF
IFEQ(T2VAL, T2MAX)
    ### Relay OFF
       WREG(A0,7,1)
    # Stop Timer T0
       MOVI(T0VAL,5001)
    # Stop Timer T1
       MOVI(T1VAL,5001)
    # Stop Timer T2
       MOVI(T2VAL,14400001)
    # Initialize Timer T4
       MOVI(T4VAL,0)
    ### Atribuição de valor a uma variavel
    MOVI(A10, 1)
ENDIF
 IFEQ(A10,A12)
   # Stop Timer T0
      MOVI(T0VAL,5001)
   # Stop Timer T1
      MOVI(T1VAL,5001)
   # Initialize Timer T3
      MOVI(T3VAL,0)
```

```
    # Initialize variable
      MOVI(A10,0)
ENDIF
 ...
$end
```

Fig. 9 Generated code.

VIII. Discussion

To measure the impact in terms of efficiency of this new application and the contribution of this work, we´ve measured the development time of a medium application to control de embedded system module for automotive industry. The scenario was an experienced programmer in assembler to develop using the regular approach, without our application and an inexperience user in terms of assembler but with knowledge about the embedded features using our proposed language. Results were encouraging, the developed time of the second operator was shorter and the adjustments number needed to achieve the same functionalities was extremely smaller. Since our language does a set of verifications, to avoid programming mistakes, code consistency was an important help for the Tiny language user. Another advantage was the generated assembly code was automatically commented using the code generator and the code from de traditional approach was sometimes unreadable and only the developer understands it.

Another noted feature was the fact that code built using the traditional approach reveals several redundancies that lead to code inefficiency. Using our code generator redundancy was avoided and code consistency improved.

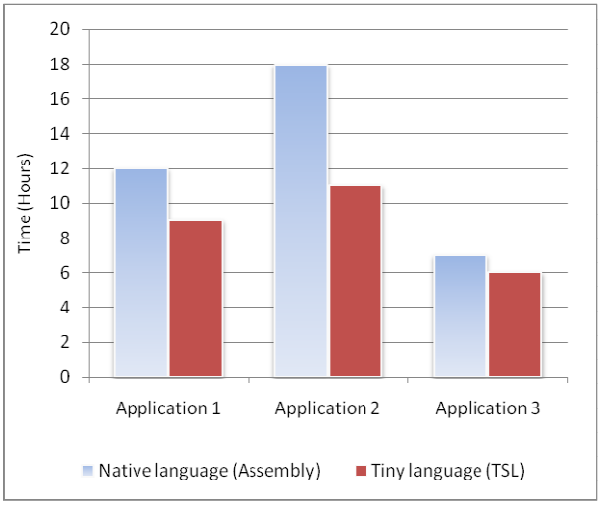

Fig. 10 Conventional methodology vs TSL based methodology development time

As stated before we measure these issues in terms of development time by two operators (one experienced in programming for the target assembly and other with experience about the module but not a regular programmer of assembler for it). For both operators we've defined a set of applications, three, for different purposes and with different complexity. In the first case (application 1) a regular application code was required, and both programmers had know-how to develop it. In the second case (application 2) consisted in the development of a larger program with several simple instructions, with a purpose not yet experienced by both programmers. The third application (application 3) had more complex instruction, namely a large number of delays. The elapsed time was measured and the results are expressed in the Figure 10.

Analyzing the results we can conclude that the developed tiny language improves the time-to-market of code for the target embedded system. The advantage of the language is most significant for long application with simple instructions. The advantage is not so clear in the specific case of complex instructions, in this case it is required that the user must also deal with assembly rules and specifications because only the skeleton is generated. However even in this case results are satisfactory. Another fact was the generated code quality, in terms of indentation and quality that exists in the generated code on an automatic basis.

IX. Conclusion and Future Work

Experiments and tests show that using the new language a short effort and design time is needed to achieve better goals. The goals are the assembly code to be uploaded for embedded systems that is used for the automotive industry. The infrastructure can be easily adapted for other similar targets. The software is running on a platform independent basis, so portability would be not a problem to other environments. Also it was defended that these tiny languages can improve significantly the development time where low lower languages are demanded in simple applications. Development tools allow on a quick and inexpensive way to develop frameworks or application that can significantly improve the development time and consequently the time-to-market of the target systems, this case an embedded systems to be used in the automotive industry.

As future work we want to implement and editor with code complete feature for our tool, to increase even more the development efficiency. This feature will allows to spare time at the editing stage of the code and to avoid code mistakes at the high level of software abstraction of this application.

As mentioned in the paper this was also and education-industry partnership and a case study, however future work is being studied also to go further and to design a graphic editor, therefore smaller blocs can be predefined and editable to reach higher abstraction and in the end a shorter development time.

References

[1] P. Ojala, "Towards a Value-Based Approach in Software Engineering," in 2nd WSEAS International Conference on Computer Engineering and Applications, Acapulco, Mexico, 2008.

[2] S. Preuer, "A Domain-Specific Language for Industrial Automation," in Conference on Software Engineering, Hamburg, Germany, 2007.

[3] D. Spinellis, "Notable design patterns for domain specific languages," *Journal of Systems and Software*, vol. 56, pp. 91-99, 2001.
[4] P. Ojala, "Experiences of Implementing a Value-Based Approach to Software Process and Product Assessment," in 2nd WSEAS International Conference on COMPUTER ENGINEERING and APPLICATIONS, Acapulco, Mexico, 2008.
[5] K. Babcicky, "Is it worthwhile to develop a new programming language?," in 12th WSEAS International Conference on Computers, Heraklion, Greece, 2008.
[6] H. Prähofer, D. Hurnaus, R. Schatz, C. Wirth, and H. Mössenböck, "Monaco: A DSL Approach for Programming Automation Systems," in Conference on Software Engineering, Munich, Germany, 2008.
[7] H. Prahofer, D. Hurnaus, and C. Doppler, "MONACO — A domain-specific language supporting hierarchical abstraction and verification of reactive control programs," in 8th IEEE International Conference on Industrial Informatics, Osaka, Japan, 2010.
[8] F. Wenzel and R.-R. Grigat, "A Framework for Developing Image Processing Algorithms with Minimal Overhead," in 5th WSEAS International Conference on SIGNAL, SPEECH and IMAGE PROCESSING, Corfu, Greece, 2005.
[9] N. H. Christensen, "Domain-specific languages in software development and the relation to partial evaluation," in *Department of Computer Science*: University of Copenhagen, 2003.
[10] P. Hudak, "Modular Domain Specific Languages and Tools," in 5th International Conference on Software Reuse, IEEE Computer Society, 1998.
[11] S. Microsystems, "Java," [Online] Available at: http://java.sun.com/, [Access date: 2009, October].
[12] E. Foundation, "Eclipse," [Online] Available at: http://www.eclipse.org/, [Access date: 2009, October].
[13] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, 2007.
[14] K. Schwaber, *Agile Project Management with Scrum (Microsoft Professional)*: Microsoft Press, 2004.
[15] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison-Wesley Professional, 1999.