DOI:10.2298/CSIS101231009K

From DCOM Interfaces to Domain-Specific Modeling Language: A Case Study on the Sequencer

Tomaž Kos[1], Tomaž Kosar[2], Jure Knez[1], and Marjan Mernik[2]

1 DEWESoft d.o.o., Gabrsko 11a, 1420 Trbovlje, Slovenia
{tomaz.kos, jure.knez}@dewesoft.si
2 University of Maribor, Faculty of Electrical Engineering and Computer Sciences,
Smetanova ulica 17, 2000 Maribor, Slovenia
{tomaz.kosar, marjan.mernik}@uni-mb.si

Abstract. Software development is a demanding process, since it involves different parties to perform a desired task. The same case applies to the development of measurement systems – measurement system producers often provide interfaces to their products, after which the customers' programming engineers use them to build software according to the instructions and requirements of domain experts from the field of data acquisition. Until recently, the customers of the measurement system DEWESoft were building measuring applications, using prefabricated DCOM objects. However, a significant amount of interaction between customers' programming engineers and measurement system producers is necessary to use DCOM objects. Therefore, a domain-specific modeling language has been developed to enable domain experts to program or model their own measurement procedures without interacting with programming engineers. In this paper, experiences gained during the shift from using the DEWESoft product as a programming library to domain-specific modeling language are provided together with the details of a Sequencer, a domain-specific modeling language for the construction of measurement procedures.

Keywords: domain-specific modeling languages, data acquisition, measurement systems.

1. Introduction

Data acquisition is the process of capturing and measuring physical data and the conversion of these results into digital form that is further manipulated by a computer program. Data acquisition systems, also called measurement systems, are used in various fields, ranging from the automotive industry to the aircraft industry, the space industry and electrical engineering. For instance, Fig.1 shows data acquisition during a flight test with the DEWESoft product. The measurements were made on a military helicopter to analyze the vibrations on the human body. The measurements in this industry, as well as

others, are quite demanding, with many repetitions on different settings. Most of the measurement procedures can be done automatically using the prepared measurement programs; however some needed to be designed manually at the time of measurement.

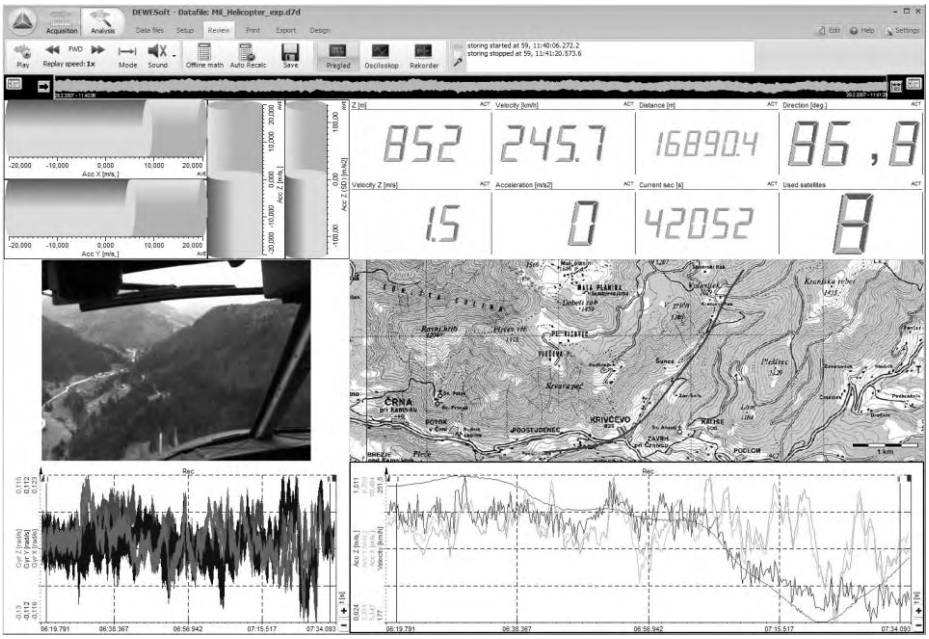

Fig. 1. Measurement system DEWESoft during helicopter vibration test

Many measurement system producers provide application programming interfaces (APIs) to use their products. Those APIs are further used by the customer's programming engineers to build software according to their specific needs. However, a customer's programming engineers do not necessarily posses knowledge about the problem domain; therefore they have to work with domain experts to prepare the desired product. In this way, prepared measurement procedures can be defined by programming engineers and further used by domain experts. As stated before, sometimes prepared measurement procedures are unsuitable and need to be repeated with slightly different settings. In that case, domain experts need to work with programming engineers to prepare another measurement procedure. Such development is time-consuming. An ideal measurement system would be, if domain experts could prepare the measuring procedures alone without the interference of programming engineers.

To support domain experts in programming their own measurement procedures and to be able to fine tune them during measurement, DEWESoft developed a domain-specific modeling language (DSML) called Sequencer. Our concrete motivation for this product was to enable domain experts to

program/model their own data acquisitions and tune them during measurements without any help from programming engineers. Domain-specific languages (DSLs) provide notations and constructs tailored toward a particular application domain [1] and therefore are suitable for domain experts that have minor programming experience and expertise in the target problem domain [2]. Compared to general-purpose languages (GPLs), like C, C++, Java, etc, DSLs are much more expressive and easy of use [3] for the domain in question. However, DSL development is often accused of having disadvantages, since it requires both domain knowledge and, in particular, language development expertise, which is rare in the programming engineering community. Therefore, it is important to present practical evidence of developing DSLs in the industry [4] and provide results regarding the end-users' satisfaction. Also, the experiences gained through the development of the Sequencer are reported in this paper.

The line between DSLs and DSMLs is often blurred and it is hard to distinguish DSLs from DSMLs. The classification often depends on personal viewpoint. Up to now, DSLs are usually textual [5, 6, 7, 8], while DSMLs further raise abstraction level, expressiveness and ease of use, since models are specified in a visual manner and coding phase is moved to specification and design phase [9, 10]. With the Sequencer, measurement procedures are possible to specify in both text and visual form. Both options are alternatives to the previous one – to construct measurement procedures with an API, which is a standard development method when using GPLs. From that prospective, in this paper some of the experience are reported regarding which notation is more popular among DEWESoft customers, as well as their feedback.

The organization of the paper is as follows. In Section 2 related work on DSMLs is presented. The design details and characteristics of the Sequencer are described in Section 3. In Section 4, development and deployment together with our experiences are presented. Finally, contributions and concluding remarks with an outline for future work are summarized in Section 5.

2. Related work

Currently, scientists and engineers in diverse areas of work, as well as end-users with specific domain expertise, require computational processes to complete a task. However, such users are typically unfamiliar with programming languages and completing their task becomes a challenge. Model-driven engineering (MDE) is an approach that provides higher levels of abstraction to allow such users to focus on the problem, rather than the specific solution on particular technology platforms. An important part of MDE is a domain-specific (modeling) language DS(M)L that fit the domain of an end-user by offering intentions, abstractions, and visualizations for domain

concepts. Many papers have been published recently on this topic and some of the most relevant ones are discussed in this section.

Jimenez, et al, show that combining a DSML with an MDE approach can enhance the quality and portability of home automation systems [11]. Most home automation systems are currently developed using proprietary low-level procedures that are platform dependent. To enhance productivity, flexibility, interoperability and end-user programming, a visual modeling language called Habitation has been designed and developed which enables the description of home automation systems using only domain concepts. The Eclipse Graphical Modeling Framework (GMF) has been used to automatically generate a graphic editor, while transformations are defined using the graph grammar approach (EMT - Eclipse Model Transformation). The main difference with our work is domains (home automation systems vs. measurement systems) and how both DSMLs have been developed. While Habitation has been developed using already existing metamodeling tools, we were not able to use them due to strong dependency on DEWESoft software.

Mathe, et al, present a Clinical Process Management Language (CPML) for capturing health treatment protocols [12]. The CPML is a formally specified visual modeling language developed using the metamodeling tool GME. The semantics have been specified using operational behavioral semantics. The semantics of the Sequencer is currently given by attribute grammars, which is used in the implementation phase, but do not enable a high level verification and analysis. In the future, our aim is to define Sequencer semantics using graph grammars.

Venigalla, et al, present a domain specific modeling language BASSML targeting spacecraft designers [13]. The BASSSML is a part of BASS, a prototype modeling tool for spacecraft systems. BASS consists of a model interpreter, which translates the captured spacecraft design models into machine-readable CSP (Communication Sequential Processes) that can be formally verified using a model checker. Using BASS, the authors show that spacecraft subsystem interfaces and interactions can be rigorously specified and analyzed. Hence, obscure subtle ambiguities and inconsistencies can be detected much earlier, thereby reducing developing costs.

Merilinna presents an end-user driven development of navigation applications for mobile phones [14]. For this purpose, a DSML was developed using the modeling environment MetaEdit+. The authors provide yet another piece of evidence that end-users, who are non-programmers, can actively participate in the development of navigation applications or develop applications completely by themselves using DSMLs within a narrow domain.

Živanov, et al, present KAG (Kiosk Application Generator), a DSL that can generate applications to be deployed on kiosks with touch-screen monitors. KAG is a nice example of DSL that upon textual specifications generates graphical-user interfaces using standard compiler generator tools (lex/yacc). Authors debate that comparing development of applications with KAG (and previous way, with general-programming languages), reduced number of programming errors and made kiosk applications development significantly faster.

A DOMMLite is the next example of DSMLs [16]. DOMMLite is used for definition of state structures of database applications. It was developed using generator framework openArchitectureWare. The domain-specific notation is defined with a metamodel supplemented by validation rules based on Check language and extensions based on Extend language that are parts of the openArchitectureWare framework. Semantics can be defined with specifications through source code generation for the supported target platforms. DOMMLite is supported with textual Eclipse editor.

DSMLs are prone to change much more often comparing to GPLs [17]. This is an emergent research area in MDE where models and modeling languages are subject to change [18]. However, in some environments, like DEWESoft, even dynamic language evolution might be necessary. In that case a system requires run-time adaptation without stopping an application. Possible solutions for adaptive DSML evolution are presented in [19, 20, 21].

3. Domain-specific modeling language Sequencer

Various implementation techniques to implement a DSL exist, such as: preprocessing, embedding, compiler/interpreter, compiler generator, extensible compiler/interpreter, and commercial off-the-shelf [1]. Of course, the language designer has to choose the most suitable implementation approach, according to the project influences [22]. In our case, the development was influenced by the fact that DS(M)L has to be included in the already-existing data acquisition software DEWESoft and that this product is developed in Object Pascal, more specifically in Delphi [23]. These limitations lead us to decide for compiler/interpreter implementation approach, where some of the compiler generator tools were used.

3.1. Construction of a textual concrete syntax

The development of DSML with the compiler/interpreter implementation approach gave us more freedom and flexibility than using other implementation approaches mentioned in [1]. In this approach, the standard compiler/interpreter techniques are used to implement a DSML. In the case of the compiler, a complete static analysis is done on the DSML program/specification. The most important advantage of this implementation approach is that the syntax is closer to the notation used by domain experts, and good error reporting. The compiler generator approach is similar to the previous one, except that some of the compiler/interpreter phases (lexical, syntax, and semantic analysis) are implemented using language development systems or so-called compiler writing tools (compiler-compilers) (e.g. Lex/Yacc [24], ANTLR [25], LISA [17], YAJCo [26]). In this manner, the implementation effort is minimized when compared to the previous approach.

Generally, the idea of a lexical analyzer is relatively simple. However, the construction and implementation of a lexical analyzer is time-consuming. Therefore, in the construction of a lexical analyzer, a compiler generator implementation approach can be used to speed up this process. In the case of the Sequencer, the help of DLex was used during the lexical analysis that generated a lexical analyzer in the programming language Delphi. With regular expressions, the formal description of the lexical analyzer was provided. Part of the DLex formal description of the Sequencer is presented in Fig. 2.

```
INTEGER                    [+-]?[0-9]+
FLOAT                      [+-]?[0-9]+(\.[0-9]+)?
BOOL                       "True"|"False"
STRING                     ['][a-zA-Z0-9.,
                           ;:%!?{|}#$&()<>=+@[\\\]/_-]*[']
COMMENT                    [/]{2,2}.*
IGNORE                     \n|\r|\r\n|" "|\t|\b
SEPARATOR                  "("|")"|","
FUNC                       "Action"  |"LoadSetup" |"If"
                           |"Loop"   |"WaitFor"   |"Delay"
                           |"AvdioVideo |"Formula"|"CustomBlock"
                           |"LaunchApplication"   |"Macro"
SPECWORDS                  "Begin"   |"End"
CONDTYPE                   "ctUser"  |"ctValue"   |"ctTrigger"
OPERATOR                   ">" | "<" | "=" | "!="
BOPERATOR                  "or" | "and"
LSTYPE                     "Static" | "Dynamic"
%%
{STRING}     begin
                  TokenList.Add(TToken.Create(yytext, tString,
                        yycolNo, yyLineNo));
             end;
...
```

Fig. 2. Lexical specification of Sequencer using DLex

The syntax and semantic analyzer has been developed independently of existing compiler generator tools. The syntax of the Sequencer was described using standard BNF notation. Part of the Sequencer's BNF is presented in Fig. 3. From the starting non-terminal NT_START, it can be seen that the reserved words (Begin, End) embody DSL statements that represent functionalities (non-terminal NT_LINE) to be performed from the measurement system DEWESoft. There are various non-terminals derived from the non-terminal NT_LINE: Action, LoadSetup, If, Loop, WaitFor, etc. For example, Action represents the basic functionality of the Sequencers' program (load project, export data, print, etc). If the load project is specified with an Action, then the hardware setup for a measurement procedure is performed. The non-terminal NT_ACTION is defined with non-terminals NT_B_ITEM (beginning parenthesis "("), NT_PACTION (action properties), and NT_E_ITEM (ending parenthesis ")" with reference to the following

functionality: non-terminal NT_LINE). The non-terminal NT_PACTION contains specific properties for the current functionality, while the non-terminal NT_PROP contains generic properties. In non-terminal NT_PROP, first terminal (#integer) presents the ID of a construct, then #string represents the text info that will be presented to the Sequencer's user interface, #boolean terminal carries information if the Sequencer will notify the end-user with text-to-speech functionality, etc.

```
NT_START ::= "Begin" NT_LINE "End"
NT_LINE  ::= "Action" NT_ACTION
           | "LoadSetup" NT_LOADSETUP
           | "If" NT_IF
           | "Loop" NT_LOOP
           | "WaitFor" NT_WAITFOR
           | "Delay" NT_DELAY
           | "AvdioVideo" NT_AVDIOVIDEO
           | "Formula" NT_FORMULA
           | "CustomBlock" NT_CUSTOM_BLOCK
           | "LaunchApplication" NT_LAUNCHAPP
           | "Macro" NT_MACRO
           | epsilon
NT_ACTION ::= NT_B_ITEM NT_PACTION NT_E_ITEM
NT_B_ITEM := "(" NT_PROP
NT_E_ITEM ::= ")" NT_LINE
NT_PROP   ::= #integer "," #string "," #boolean "," #integer ","
              #integer "," #integer
...
```

Fig. 3. Syntax specification of Sequencer

```
function TSeqParser.NT_LINE(Lexer : TLexer; Group :
                                        TSeqGroup) : Boolean;
var
  Item : TSeqItem;
  I : Integer;
begin
  Result := False;
  Item := nil;
  if (Lexer.CurrentToken.AType = tFunc) then
  begin
    if (Lexer.CurrentToken.Lexem = 'Action') then
    begin
      Lexer.NextToken;
      Item := Group.SeqItems.AddNewItem(it_Action);
      Result := NT_ACTION(Lexer, Group, Item);
    end
    else if (Lexer.CurrentToken.Lexem = 'LoadSetup') then
    begin
...
```

Fig. 4. Semantic of the Sequencer's non-terminal NT_LINE

The semantics of the Sequencer is described using attribute grammars from which a compiler is automatically generated. In the semantic part,

attributes carry the values of actions defined in a DSL program and are responsible for calling functionalities from DEWESoft environment. Fig. 4 presents the part of the Pascal code for production NT_LINE. First, the token has to be checked which should be “tFunc” and the lexem should be “Action”. After that the lexical analyzer goes to the next token and to the next production which is in our case NT_ACTION.

The language processing effort is usually divided into syntax and semantic parts. In the syntax, the lexical analyzer and syntax analyzer size has been checked and 2,787 lines of code (LOC) have been generated or written. The semantic part of a code that contains all library calls to the DEWESoft framework contains 5,102 LOC. All together, the Sequencer DSL contains 7,889 LOC, which was developed in six engineer months. Since the Sequencer’s first release, new features and updates were occasionally introduced over the following six months, which were not counted as development time.

3.2. Transformations in the Sequencer

```
- <Sequences RootID="27">
  - <Sequence>
      <ID>1</ID>
      <Type>1</Type>
      <TextInfo>Load Setup</TextInfo>
      <DisplayRow>3</DisplayRow>
      <DisplayCol>1</DisplayCol>
      <NextSeqItem1_ID>28</NextSeqItem1_ID>
      <FileName>AccTest.d7s</FileName>
    </Sequence>
  - <Sequence>
      <ID>3</ID>
      <Type>3</Type>
      <TextInfo>Enter file details</TextInfo>
      <DisplayRow>11</DisplayRow>
      <DisplayCol>1</DisplayCol>
      <NextSeqItem1_ID>4</NextSeqItem1_ID>
    - <Condition Index="0">
        <Value0>"</Value0>
        <Value1>0</Value1>
      </Condition>
    </Sequence>
  - <Sequence>
```

Fig. 5. Sequencer’s code in XML

Transformations in the Sequencer are an important part of the tool. Their purpose is to transform programs into execution code that is further executed in the Sequencer’s framework. In the case of the Sequencer, the

transformation occurs when a program is transformed into another presentation, execution model or vice versa. The transformation is carried out according to the selected initial and final model.

All transformations are in the group of exogenous transformations [27], because a model could never be transformed in the same model. Transformations enable one to change programs from XML to text or visual notation without any loss.

XML is also used in the Sequencer as an export and saving format (Fig. 5). Execution code in transformed into XML and with that feature, the portability and ability to exchange sequences between end-users and customers is supported.

3.3. Construction of visual concrete syntax

Beside textual notation, also visual notation has been developed for the Sequencer. For this purpose metamodels are often used. Usually, the metamodel is constructed using a standalone metamodeling tool [28, 29], a specialized software for the construction of DSMLs. However, DSML can have an implicit metamodel and in the case of the Sequencer, it was decided to prepare a fixed metamodel where the models were transferred to the execution model. In the Sequencer's metamodel the following domain concepts have been defined:

- a set of classes,
- associated attributes for each class,
- the relationship between classes, and
- constraints between classes.

Regarding the constraints in the Sequencer: there are no constraints on relations in the modeling language – each class can be connected to the others.

For each class a building block (concrete syntax) has been defined. In general, building blocks are separated into shapes and links. Each shape has a unique presentation in the form of a color, size and shape type (rectangle, diamond, ellipse, etc.). In the Sequencer, links have a unified form (line with arrow). Each shape belongs to exactly one building block and the link corresponds to a relationship. Each building block represents an action from a measurement system. Actions start their execution in the initial building block (marked with a circle) and continue to the next building block that is connected with the link.

Building blocks also contain local and global variables (that represent channels in DEWESoft). Their purpose is to store specific values in measurements. History is available for those variables and this is further used to plot graphs after the measurement is finished.

Regarding the Sequencer’s visual notation, a custom block has been introduced, that embodies several building blocks in a single one. When there are a lot of building blocks in a measurement procedure, a model can become

unmanageable. With custom blocks, larger sequences can become more readable.

Nowadays, most measurement software is designed for capturing, storing and analyzing the measured data and do not allow the manual construction of the measuring process. They provide customizations, where you can tune the measurement procedure with only a few options. With the Sequencer, DEWESoft has decided to step forward and has developed a powerful DSML for the purpose of measurement procedures.

4. Results

In this section, the experience of using the Sequencer is discussed. Firstly, the Sequencer DSML is compared to other selected DSLs to observe its size and complexity. Then, Sequencer programs are compared to previous applications developed with DCOM objects. In the end, some experiences are reported from the end-users and numbers are given about how many customers are already using the Sequencer; the new feature of a DEWESoft product.

4.1. Sequencer complexity

From the language developer's point of view, it is worthwhile to observe the size of a language. The easiest way to do this is to compare it to other languages. It has been decided to compare just the Sequencers' textual notation and the following DSLs were chosen for comparison with Sequencer:

- Production Grammars (PG) for software testing [6],
- A DSL that allows experimentation for the different regulation of traffic lights (RoTL) and supports the domain-specific analysis of junctions [7],
- Context-Free Design Grammar (CFDG)[1], designed for generating pictures from specifications,
- GAL, a well-known DSL used to describe video device drivers [8].

One can get grammar examples with various compiler tools; however these are unsuitable for a comparison with the ones used in practice, since they are usually small owing to the fact that their value is in learning a specific tool notation. Our aim was to compare the Sequencer's grammar with the ones already applied in practice. In existing literature, those grammars are often partially presented, since they are usually too long to fit in the paper. Therefore, the above grammars were selected for comparison since they are used in practice and a full grammar was available to the authors of this paper.

[1] Context-Free Design Grammar, available at http://www.chriscoyne.com/cfdg/index.php

The size of a DSL can be compared to others using grammar metrics [30, 31]. In [30] grammar metrics are divided into size and structural metrics. For the purpose of our comparison we took the following size metrics:

- term – number of terminals,
- var – number of non-terminals,
- avs – average of right hand side size,
- mcc – McCabe cyclomatic complexity, and
- hal – Halstead effort.

Let us briefly discuss the above-mentioned metrics. A greater maintenance is expected if a grammar has a large number of non-terminals (var). The metrics mcc measure the number of alternatives for grammars' non-terminals. A high value indicates a potentially larger effort for grammar testing and a greater potential for parsing conflicts. A big avs value usually means that grammar is less readable. The Halstead effort metric (hal) estimates the effort required to understand the grammar. Grammar metric comparisons between the Sequencer and selected DSLs were obtained by the tool gMetrics [31] (Table 1). From the results of the size metrics, it can be concluded that the Sequencer is comparable to many of the selected DSLs. Of course, DSL complexity depends on the domain and can be much larger than other DSLs (observe GAL results on grammar metrics in Table 1).

Table 1. Comparison of Sequencer with other DSLs

DSL	TERM	VAR	AVS	MCC	HAL
Sequencer	24	31	4.61	0.52	16.21
PG	10	5	3.80	1	0.89
RoTL	23	12	4.83	0.5	3.89
CFDG	24	13	6	2.38	6.57
GAL	71	74	3.88	1.20	33.36

4.2. Comparison of DCOM applications with the Sequencer's programs

The advantage of Sequencer over application development with DCOM objects can be observed when comparing a program from Fig. 6 with the DCOM application in Fig. 7. The advantages compared to APIs are obvious in respect to the clarity and understandability of the code.

Both programs (Fig. 6 and 7) describe the procedure (sequence) which is prepared to guide one through the entire car acceleration test maneuver. Besides the acceleration test, in the automotive industry, different measurements are applied to cars, like brakes, tires, a fuel consumption test, etc. The sequence in programs (Fig. 6 and 7) starts with the project and setup file load and the setup screen is shown. The start and stop speed can be set here. The next step is file details. Here the end-user has to set the file name and some test details (car type, driver, place, road surface, etc.). After this, the end-user starts driving. When reaching certain conditions (speed,

temperature, pressure, distance) that are necessary to perform the acceleration test, the system advises the user to accelerate to the target speed. During the measurement process, the end-user can observe vehicle speed, vehicle acceleration, acceleration distance, temperature, etc. When the measurement is finished the end-user has the option of repeating the test or continuing to analyze and then printing out the stored data.

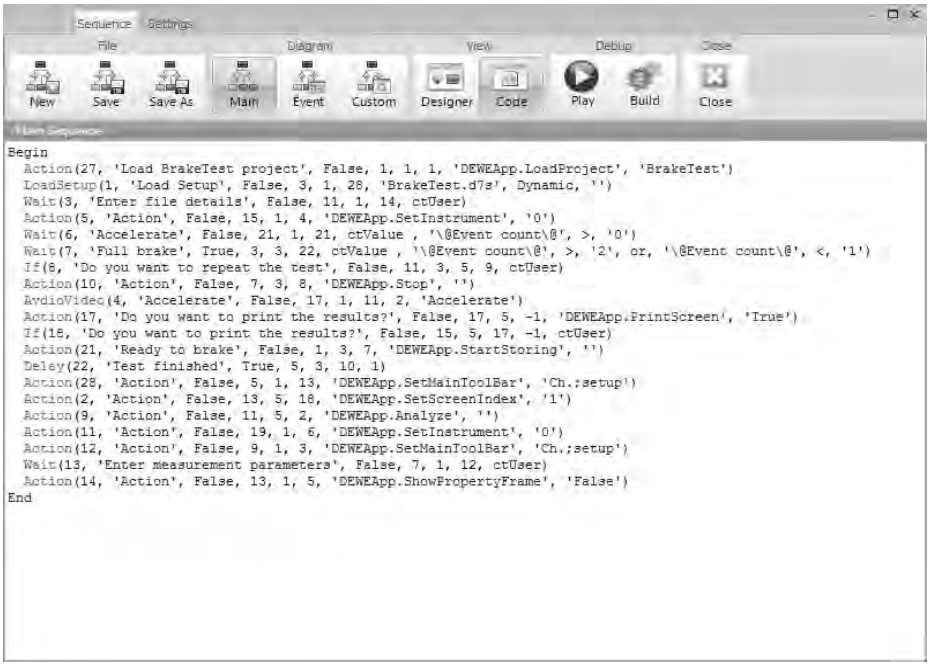

Fig. 6. Sequencer program in textual notation

```
unit Unit2;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls,
  Forms, Dialogs, AdvGlowButton, AdvToolBar, StdCtrls,
  AdvCaptionPanelUnit, DEWEsoft_TLB, ExtCtrls;
const
  bt_Yes = 1;
  bt_No = 2;
  bt_Continue = 4;
  SVSFlgAsync = $00000001;
type
  TForm2 = class(TForm)
    SequencerControlPanel: TAdvCaptionPanel;
    SequenceInfoLabel: TLabel;
    SequenceSeparator: TAdvToolBarSeparator;
    SequencePlayButton: TAdvGlowButton;
```

```
    ...
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Panel1Resize(Sender: TObject);
    ...
  private
    DeweApp : App;
    CurrState : Integer;
    oVoice : OLEVariant;//TTS
    procedure KillProcess(const ProcName : string);
    procedure SetHeader(Caption : string; Buttons : Integer);
  public
  end;
var
  Form2: TForm2;
implementation
uses
  Registry, TlHelp32, ComObj;
{$R *.dfm}
procedure TForm2.Panel1Resize(Sender: TObject);
begin
  if Assigned(DeweApp) then
  begin
    DeweApp.Left := 0;
    DeweApp.Top := 0;
       DeweApp.Width := panel1.Width;
              DeweApp.Height := panel1.Height;
  end;

end;
...
```

Fig. 7. DCOM application

Table 2. Comparison of Sequencer applications with DCOM applications in LOC

DSL	DCOM application	Sequencer	Ratio
Application 1	308	22	14
Application 2	298	15	19,86
Application 3	301	23	13,09
Application 4	280	20	14
Application 5	325	15	21,66

Another advantage can be observed if the Sequencer programs are compared with the DCOM application with the number of lines of code. In Table 2, the size of code (LOC) is presented for five different applications developed with Sequencer and DCOM objects. All Sequencer programs and DCOM applications have the same functionality. Table 2 confirms the advantage of Sequencer compared to the API solution (observe the ratio column in Table 2), since the Sequencer programs were at least 13 times shorter than the same DCOM applications. Similar productivity increase has

been reported also elsewhere (e.g., [28]). Note, that applications in Table 2 are case study problems.

4.3. Customers' experiences

The DEWESoft product has already been successfully applied to the car industry. For example, the DEWESoft product is used by TÜV, an independent German consultant organization that validates the safety of products, like motor vehicles. Also, DEWESoft's measurement units (together with its software solution) are used in aviation, construction, electric and even aerospace industry. NASA awarded the DEWESoft product as "Product of the year" in 2009. From Table 3, it can be observed that DEWESoft has over 500 end-users who are using measurement systems for their specific measurements. Also, there are over 40 programming engineers who are using our DCOM objects to develop measurement procedures for their end-users.

Since January 2010, when Sequencer was released with DEWESoft ver. 7, over 150 end-users have already used the measurement procedures with the Sequencer. More than 30 domain experts are already developing sequences with the new feature of DEWESoft.

The real value of the Sequencer can be found in the last column of Table 3, which shows how many new domain experts have started using DEWESoft since the product became easier to use.

Table 3. DEWESoft customers

DCOM application end-users	DCOM programmers	Sequencer end-users	Sequencer domain experts	New domain experts on Sequencer
500	40	150	30	20

4.4. Sequencers' textual vs. visual notation

Both textual as well as visual concrete syntaxes have implemented the exact same functionalities and can therefore be transformed from one notation to another, as described in subsection 3.3. From the Sequencer developers' point of view, both notations are available to customers of the measurement system DEWESoft and they were not encouraged to use either of them.

Fig. 8 presents the Sequencers' modeling environment. The building blocks are on the left side of the environment. On the right side of the environment there are variables that can be selected for each individual building block. In the middle of the environment, the end-user can construct the measuring sequence with visual notation. Visual building blocks are used with "drag and drop" functionality. The Sequencer leads the end-user through a

measurement procedure using static analysis, thereby reducing the possibility of human error and increasing the efficiency of the test itself.

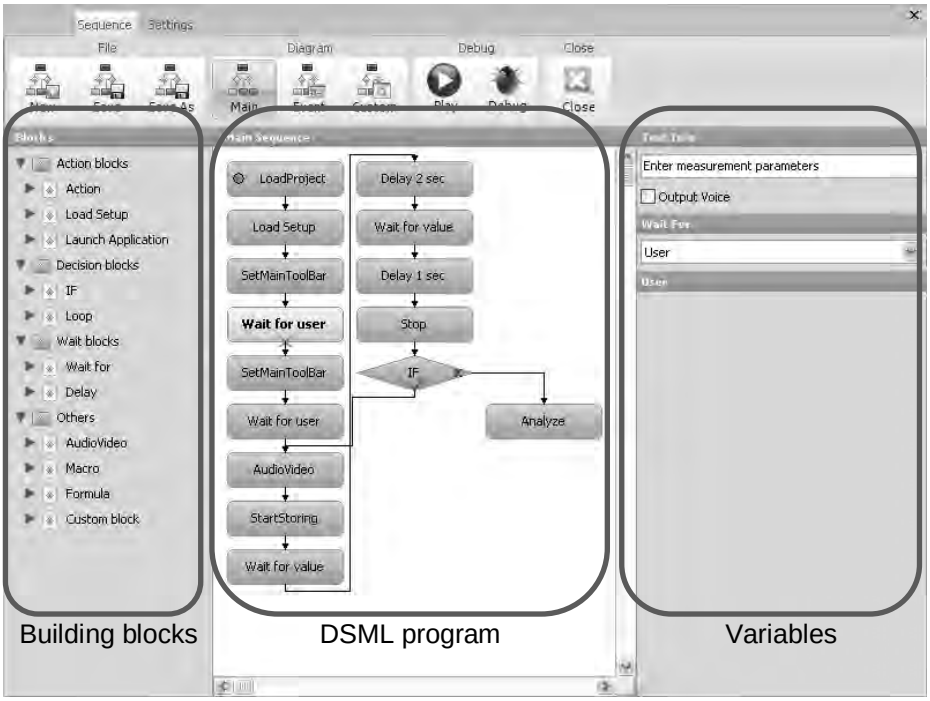

Fig. 8. Sequencer's modeling environment

Studying the Sequencers' domain experts revealed that most of them are using this visual notation rather than the text version of the Sequencer. The most probable explanation for this lies in the abstraction level of both notations. Also, there appears to be a general opinion that in order to use textual notation, the end-user needs a certain degree of programming experience. Both reasons, probably influenced end-users to prefer using the visual version of the Sequencer.

5. Conclusion and future work

The purpose of the Sequencer was to enable the easier construction of measurement procedures inside the measurement system DEWESoft. The main goal of the Sequencer is to push the development of the application from using DCOM objects to a specialized tool that enables domain experts to develop measurement sequences efficiently in a simple manner, without the need of support from programming engineers. Sequences can be developed in a textual or visual mode, which are customized for application development

in the measurement domain. In this paper, the experiences in the development of Sequencer as well as experience with end-users were presented. According to the opinion of domain experts, the construction of the Sequencer has been a good step in simplifying complicated measurement development in many different fields.

From a usability point of view, the Sequencer's next feature is to record a sequence execution and save it in text format. In this manner, sequences can be analyzed in time to see more details. Currently, the system enables users to study the final results of the measurement test. From a DSML point of view, the next development effort will be to support domain experts with domain-specific debugging facilities similar to one presented by Wu, et al [32].

DS(M)Ls are promising for the future development of software, since current software development, centered on GPLs, is becoming more and more complex and software customization usually involves a larger effort on the part of programming engineers. On the other hand, DSLs enable domain experts to program and make changes in software and with that they can quicken development and reduce maintenance costs.

References

1. Mernik M., Heering J., Sloane A.M.: When and how to develop domain-specific languages. ACM Computing Surveys, Vol. 37, No. 4, 316–344. (2005)
2. Sprinkle J., Mernik M., Tolvanen J.-P., Spinellis D.: Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26, No. 4, 15-18. (2009)
3. Kosar T., Oliveira N., Mernik M., Varanda Pereira M.J., Črepinšek M., da Cruz D., Henriques P.R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
4. Ferreira E., Paulo R., da Cruz D., Henriques P.R.: Integration of the ST Language in a Model-Based Engineering Environment for Control Systems – An Approach for Compiler Implementation. Computer Science and Information Systems, Vol. 5, No. 2, 87-101. (2008)
5. Arora R., Bangalore P., Mernik M.: Raising the level of abstraction for developing message passing applications, The Journal of Supercomputing, (2010). Accepted for publication, doi: 10.1007/s11227-010-0490-3
6. Sirer E. G., Bershad B.N.: Using production grammars in software testing. In: Proceedings of the 2nd Conference on Domain-Specific Languages, pages 1-14. USENIX Association (1999)
7. Mauw S., Wiersma W.T., Willemse T.A.C.: Language-driven system design. International Journal of Software Engineering and Knowledge Engineering, Vol. 6, No. 14, 625-664. (2004)
8. Thibault S., Marlet R., Consel C.: Domain-specific languages: from design to implementation - application to video device drivers generation. IEEE Transactions on Software Engineering, Vol. 25, No. 3, 363-377. (1999)
9. Schmidt C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer, Vol. 39, No. 2, 25-31. (2006)

10. Gray J., Tolvanen J.-P., Kelly S., Gokhale A., Neema S., Sprinkle J.: Domain-Specific Modeling. Handbook of Dynamic System Modeling. Boca Raton, Florida: CRC Press (2007)
11. Jimenez M., Rosique F., Sanchez P., Alvarez B., Iborra A.: Habitation: A Domain-Specific Language for Home Automation. IEEE Software, Vol. 26, No. 4, 30-38. (2009)
12. Mathe J., Ledeczi A., Nadas A., Sztipanovits J., Martin J., Weavind l., Miller A., Miller P., Maron D.: A Model-Integrated, Guideline-Driven, Clinical Decision Support System. IEEE Software, Vol. 26, No. 4, 54-61. (2009)
13. Venigalla S., Eames B., McInnes A.: A Domain Specific Design Tool for Spacecraft System Behavior. In DSM, Nashvile, TN (2008)
14. Merilinna J.: Domain-Specific Modelling Language for Navigation Applications on S60 Mobile Phones. In DSM Nashvile, TN (2008)
15. Živanov Ž., Rakić P., Hajduković M.: Using Code Generation Approach in Developing Kiosk Applications. Computer Science and Information Systems, Vol. 5, No. 1, 41-59. (2008)
16. Dejanović I., Milosavljević G., Perišić B., Tumbas M.: A Domain-Specific Language for Defining Static Structure of Database Applications. Computer Science and Information Systems, Vol. 7, No. 3, 409-440. (2010)
17. Mernik M., Žumer V.: Incremental programming language development. Computer Languages, Systems & Structures, Vol. 31, No. 1, 1-16. (2005)
18. Meyers B., Vangheluwe H.: A framework for evolution of modeling languages, Science of Computer Programming, doi:10.1016/j.scico.2011.01.002. (2011)
19. Kollár J., Václavík P., Wassermann Ľ.: Data driven Executable Language Model. In: Proceedings of the International Multiconference on Computer Science and Information Technology, pages 667-675, Polish Information Processing Society (2009)
20. Forgáč M., Kollár J.: Adaptive Approach for Language Modification. Journal of Computer Science and Control Systems, Vol. 2, No. 1, 9-12. (2009)
21. Kollár J., Forgáč M.: Combined Approach to Program an Language Evolution. Computing and Informatics, Vol.29, 1103-1116. (2010)
22. Kosar T., Martínez López P.E., Barrientos P.A., Mernik M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, Vol. 50, No. 5, 390-405. (2008)
23. Cantù M.: Mastering Delphi 7. Sybex Inc. Alameda, CA. 2003
24. Levine J. R., Mason T., Brown D.: Lex & Yacc. O'Reilly, Cambridge, MA. 1992
25. Parr T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. 2007
26. Porubän J., Forgáč M., Sabo M., Běhálek M.: Annotation Based Parser Generator. Computer Science and Information Systems, Vol. 7, No. 2, 291-307. (2010)
27. Bézivin J.: From Object-Composition to Model-Transformation with the MDA. In: TOOLS-USA'2001, Santa Barbara, USA. (2001)
28. Kelly S., Tolvanen J.-P.: Domain-Specific Modeling Enabling Full Code Generation. John Wiley & Sons, Inc. (2008)
29. Buchwalder O.: MEtaGile: An Agile Domain-Specific Modeling Environment. (2008)
30. Power, J. F. and Malloy, B. A.: A metrics suite for grammar-based software. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 16, No. 6, 405–426. (2004)
31. Črepinšek M., Kosar T., Mernik M., Cervelle J., Forax R., Roussel G.: On Automata and Language Based Grammar Metrics. Journal on Computer Science and Information Systems, Vol. 7, No. 2, 310-329. (2010)

32. Wu H., Gray J. G., Mernik M.: Grammar-driven generation of domain-specific language debuggers. Software practice and experience, Vol. 38, No. 10, 1073-1103. (2008)

Tomaž Kos has graduated at the Faculty of Electrical Engineering and Computer Science, University of Maribor, in 2009. Currently, he is a PhD student and he works for DEWESoft company as a researcher. His main research interests include programming languages, domain-specific (modeling) languages, testing, data acquisition, and measurement systems.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Jure Knez received the M.Sc. adn Ph.D degrees in mechanical engineering from the University of Ljubljana in 1999 and 2002 respectively. He is currently employed as CTO of Dewesoft d.o.o. in Trbovlje, Slovenia. The company is developing the software and measurement instrument widely used in automotive, aerospace, industrial, power distribution and civil engineering applications.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modelling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Received: December 31, 2010; Accepted: March 24, 2011.