Debugging measurement systems using a domain-specific modeling language

Tomaž Kosar*, Marjan Mernik*, Jeff Gray°, Tomaž Kos+

*University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, 2000 Maribor, Slovenia
+DEWESoft d.o.o., Gabrsko 11a, 1420 Trbovlje, Slovenia
°University of Alabama, Department of Computer Science, Tuscaloosa, AL 35487, USA

Abstract

Capturing physical data in the context of measurement systems is a demanding process that often requires many repetitions with different settings. To assist in this activity, a domain-specific modeling language (DSML) called Sequencer has been developed to enable the improved definition of measurement procedures. With Sequencer, the level of abstraction has been raised and sophisticated changes in measurement procedures are now enabled. Although there are numerous DSMLs like Sequencer in the existing literature, there are some obstacles working against the more widespread adoption of DSMLs in practice. One challenge is the lack of supporting tools for DSMLs, which would improve the capabilities of end-users of such languages. For instance, support for debugging a model expressed in a DSML is often neglected. The lack of a debugger at the proper abstraction level limits the domain experts in discovering and locating bugs in a model. In this paper, Sequencer is presented together with debugging facilities (called Ladybird) that are integrated in a modeling environment. Ladybird supports different execution modes (e.g., steps, breakpoints, animations, variable views, and stack traces) that can be helpful during the debugging of a model. Ladybird's primary contribution is in showing the value of error detection in complicated industrial environments, such as data acquisition in automotive testing. The paper contributes to a discussion of the implementation details of DSML debugging facilities and how such a debugger can be reused to support domains other than the measurement context of Sequencer.

Keywords: Debugging aid, domain-specific modeling languages, graphical environments, usage experience

1. Introduction

Domain-specific languages (DSLs) [1], [2], [3] allow domain experts to play a vital role in the software development process. Empirical evidence has shown that productivity increases with DSL adoption when compared with the traditional code-driven software development process [4], [5] that uses general-purpose languages (GPLs), such as Java or C++. The adoption of a DSL raises the level of abstraction [6], [7] and connects the concepts from the problem and solution domains [8], [9]. Domain experts, who have skills in the problem domain, but may not have formal training in computer science, can write their own domain-specific programs to solve a specific need in their domain [10]. DSLs can be further sub-divided into specification, modeling, and programming languages [11]. In this paper, we focus on domain-specific modeling languages (DSMLs) [12], [13], which often use a visual notation rather than textual representation and remain more expressive at a higher abstraction level than GPLs.

In measurement systems, both mechanical equipment and measurement settings have to be tested from various points of view. If traditional software tools based on GPLs are used, this process can be simplified by using the prepared test procedures to speed up the testing process and to analyze the measurement results. However, prepared tests are often insufficient and tests need to be changed, or even developed from scratch. Therefore, it would be a significant contribution to support domain experts with the ability to model measurement procedures on their own. This can be achieved by developing an appropriate DSML, which is very suitable for the construction of measurement systems [14], where physical data are captured and the conversion of these results into a digital form is performed [15]. To improve flexibility and productivity, DEWESoft [16] developed a DSML called Sequencer [17], which enables domain experts to model and evolve their own measurement procedures without any help from programmers. In Sequencer, the measurement procedure can be constructed in a textual or visual manner. To the best of our knowledge, specialized measurement systems [18], [19] do not allow the construction of measurements to such an extent. Also, existing tools are adjusted specifically to the type of test and are limited in their flexibility and usefulness. These tools (when compared with Sequencer) are limited to one type of hardware vendor, while DEWESoft supports many different types of hardware from various vendors. Moreover, with existing tools it is impossible to hide the unimportant details of the measurements (e.g., Sequencer al-

lows customizations for specific tests, while specialized measurement systems usually support limited flexibility of displayed data during measurements). DEWESoft addresses these shortcomings in Sequencer, where end-users can adjust the measurement and control procedures, while tailoring them to their specific needs by writing an additional sequence (measurement procedure).

However, sequences may become complex such that domain experts face several challenges when trying to detect bugs in the models. Bugs may occur due to specification errors (e.g., semantic errors) or measurement errors (e.g., hardware malfunction). To facilitate sequence construction, the domain expert must be empowered with dedicated tools to improve measurement procedures. Usually, DSML tools other than model compilers (i.e., transformations from models to some other artifact) are most often not available. However, the utility of a DSML is seriously diminished if the supporting tools (e.g., a debugger) needed by a software developer are not available. In this paper, we describe debugging features in a tool called Ladybird, which is integrated in the modeling environment Sequencer [17]. The debugging facilities of Ladybird (e.g., execution modes, steps, breakpoints, animations, print statements, variable view, and stack traces) enable end-users to simultaneously watch multiple models, and during the execution, monitor and alter the state of a running model. Sequencer, as well as features of Ladybird, has been applied to automotive domain[1], where the quality of the car and its parts are subjected to testing procedures.

The remainder of this paper is organized as follows. Section 2 describes related work on DSMLs, debuggers, and measurement systems. Section 3 highlights the DEWESoft system and gives some details about specific domains where the measurement system has been used. Section 4 explains the architecture of Sequencer and introduces the details of the domain-specific modeling language and the modeling tool. Ladybird, Sequencer's debugger, is discussed in Section 5 and implementation details of debugging support are presented in Section 6. A demonstration on a real case scenario is illustrated in Section 7. Discussion follows in Section 8. Finally, Section 9 provides concluding remarks and summarizes the main features of the Ladybird model debugger for the DEWESoft measurement system.

[1]Sequencer has been applied successfully in the automotive industry (e.g., General Motors).

2. Related work

This section provides an overview of related work in the area of model debuggers and measurement systems.

2.1. An Overview of Model Debuggers

The benefits of using a DSML in software engineering are still not realized fully because the software development lifecycle using DSMLs is often not supported by the appropriate tools. As observed in the case of functional languages, there were several factors that led to the resistance of functional languages in mainstream development: the lack of debuggers and profilers, inadequate support by Integrated Development Environments (IDEs), and poor interoperability [20]. These same factors can all be considered contributing factors for the software industry's resistance to DSMLs. Hence, it is crucial that more work is devoted to DSML tooling [21]. In particular, we are interested in DSML debugging tools. We believe that in the future, DSML debuggers will be generated automatically from metamodels. However, we need to gather enough experience and knowledge from the development of specific DSML debuggers in order to generalize the case to support automatic generation. Our work in this paper is a step toward that direction. Despite the fact that metamodeling tools (e.g., MetaEdit+ [12], [22], GME [23], EMF [24]) do not automatically generate DSML debuggers, we will briefly describe GME tool and its capabilities. The situation is better for textual DSLs, where some tools have integrated DSL debuggers (e.g., MPS [25]).

The GME [23] is a metamodeling tool that is similar in purpose to MetaEdit+ [12], [22], but differs in its specific use. The metamodel in GME is depicted with a UML class diagram [26] showing elements of the DSML and how they can be associated with each other. Numerous DSMLs have been built using GME (e.g., POSAML [27], PICML, CQML, CUTS [28]). It is important to note that while these tools (e.g., MetaEdit+, GME) are advantageous in constructing DSMLs, they do not support model debugging at a level of abstraction that can be used by most domain experts. Hence, debugging can be performed at the code level only, but not at the model level. At most, one can use the modeling tool API for viewing and manipulating a model's internal representation, which is not sufficient for most end-users.

Developing a DSL debugger from scratch can be very expensive. Therefore, Wu et al. [29] proposed a grammar-driven technique to build a DSL

debugger, where the debugger could be generated automatically with minimal additional effort by reusing an existing GPL debugger. However, their approach is applicable only when a DSL is implemented using source-to-source translation of a textual language, where a line of DSL code is consecutively translated into many lines of GPL code. By keeping track of the DSL code to GPL code translation, a GPL debugger can be reused, but debugger actions like "step into" and "step over" have to be reimplemented. Wu et al.'s framework has been extended to generate DSL testing tools automatically, but with similar limitations [30].

In the case of DSML debugging, relevant works are rare. Mannadiar and Vangheluwe proposed a conceptual mapping of debugging concepts from programming languages to DSMLs [31]. Language primitives (e.g., print statements, assertions, and exceptions) and debugger primitives (e.g., execution modes, steps, runtime variable I/O, breakpoints, jumps, and stack traces), which are commonly found debugging facilities in programming languages were mapped into a DSML debugger, forming a starting point for DSML debugger development. As a proof of concept, Mannadiar and Vangheluwe prototyped debugging concepts into a successor of the ATOM^3 tool [32]. They also pointed out two different facets of DSML debugging: the debugging of model transformations, and the debugging of domain-specific models. In our work, special attention will be given to the latter, but initial work in the former is represented by Hibberd et al. [33]. In our work, debugging concepts from [31] have been used in the implementation of the Ladybird debugging tool described in this paper.

Blunk et al. [34] presented an approach for modeling debuggers for a DSML. Their approach requires a metamodel-based description of the abstract syntax of the language (e.g., defined in the Eclipse Modeling Framework - EMF [35]). Debugging is then described with operational semantics at the metamodeling level, where possible runtime states are modeled as part of a DSL metamodel and transitions are defined as a model-to-model transformation. Blunk et al. demonstrated their approach on breakpoints by stepping over lines of a DSL designed for voice control. Although this approach presents modeling of the debugger at the level of the semantic definition of the DSML, our approach is more similar to EDF (Eclipse Debugging Framework) [36] where a DSL debugger is implemented manually. Similar to our approach, execution semantics is transformed to the base language. However, EDF provides a more generic solution than Sequencer because it defines a set of interfaces, which the language engineer must implement to develop

the debugger for an arbitrary DSL. These interfaces forward user interactions to concrete implementations that display information in the user-interface.

2.2. Summary of Related Work in Measurement Systems

We previously published a paper about our preliminary work [37], which provided a formulation of the challenges and an outline of a solution in the form of the envisioned debugging facilities in Sequencer (this early paper does not present the ideas behind debugging facilities or implementation details). Our current work described in this paper is an extension of the earlier conference paper, containing more in-depth discussion of debugging functionalities. A core contribution in the current paper is the presentation of Ladybird's architecture and implementation details about steps, breakpoints, animations, variable views, and stack traces as they relate to concepts of debugging the abstractions in a model.

Sequencer was previously published in [14], which introduced the design, implementation and usage of the DSML for a data acquisition system. Details on chosen terminology and concepts from the domain of measurement systems are given in that paper. Also, other parts of the DSML (e.g., runtime environment and domain framework) are described in [14]. Because this current paper highlights the debugging facilities in Sequencer, in this paper we only introduce those details of Sequencer that are needed to understand the implementation of Ladybird. More details about the complete DSML can be found in [14].

National Instrument's LabVIEW [38] is a graphical programming environment for construction of a system for data acquisitions. Their graphical programming language is based on the dataflow concept. The language is called G and the written programs are called virtual instruments. The basic building blocks in G are: a block diagram, a front panel and a connector panel. LabVIEW's graphical language is very flexible for defining a measurement procedure, but very difficult to start with for someone without development knowledge. Labview's data acquisition system has matured over the last 30 years. As claimed in [38], graphical programming languages have evolved toward a "general-purpose programming environment." This offers several advantages because it allows the environment to be used in various domains (e.g., automation environments, robotics, avionics). However, it is consequently necessary for the user to understand the details of the whole measurement system. The philosophy in DEWESoft's product is a bit different - the user focus of Sequencer is on the measurement procedure and not

on the construction of the measurement system. The users of Sequencer are allowed to construct a measurement procedure without any additional effort in a manner that is more intuitive as it relates to the users domain.

3. The DEWESoft measuring system

DEWESoft [16] is a modern measurement product that contains different modules like counters, video, vehicle data buses (e.g., CAN - Controller Area Network and FlexRay [39]), GPS (Global Positioning System) and digital telemetry interfaces in the aerospace industry [40]. DEWESoft has the ability to connect the measurement modules of different vendors. Another powerful feature of DEWESoft is that the captured data between the measurement modules and the measurement client flows via Ethernet networks. DEWESoft combines all the advantages of a desktop computer (e.g., visualizations, data storage, processing) and is a significant alternative to traditional measurement instruments and specialized products. DEWESoft is used in various industries (e.g., automotive, aviation, construction, electrical and aerospace [41]).

3.1. Measurements in the automotive industry

Measurements in the automotive industry have different purposes, such as satisfaction of different regulations related to testing safety of a vehicle. For instance, with DEWESoft one can test technologies such as ABS (Anti-lock Brake System), ESP (Electronic Stability Control), and ACC (Autonomous Cruise Control). Those technologies require deep testing before the automobile can be sent to the market. Such systems need to be tested in different conditions (e.g., weather) and on various polygons (e.g., different ground basis). Therefore, the process of validating a vehicle is a cycle that contains many repetitions. An analysis of measured signals with special equipment supplies vehicle manufacturers with useful information and thereby allows them to fine-tune their products. In addition to different vehicle manufacturers, measurement systems are used by certification authorities that test the vehicles on the basis of different standards and regulations (e.g., TÜV[2] – Technischer Überwachungs-Verein – the Technical Inspection Association).

The most frequently used tests in the automotive industry are connected with automotive safety [42]. However, safety is not the only issue that is

[2]http://www.tuv.com/us/en/index.html

tested during the development of a vehicle. There are different regulations that the product needs to satisfy, for instance noise emissions. Community noise regulations put stringent requirements on road vehicle noise emissions (in the European Union, noise emissions of four-wheel vehicles are addressed by directive $70/157/EEC$ [43]). Regulations cover the sound emitted by moving/stationary vehicles, engines and other vehicle parts (e.g., tires, exhaust) and also noise inside the vehicle (e.g., air conditioning). Many different levels of vehicle manufacturing have an interest in noise tests. Across the production chain, noise tests are also of interest to car component manufacturers. In a vehicle, every component is a potential source of noise.

3.1.1. "Pass-by" noise test

During presentation of Sequencer's debugging features, we will use the "pass-by" noise test, defined by international standards ISO362. In this test, the vehicle being driven is tested by the noise the vehicle provides to the environment. During the measurements with DEWESoft, the test driver can observe velocity, revolutions per minute, temperature and the sound levels as shown in Fig. 1. The pass-by noise test is started when certain values are satisfied (e.g., entry speed, acceleration, gust wind speed, air temperature). At the end of the test with DEWESoft's measurement system, required parameters are calculated (e.g., dBA levels, FFT, histogram, order analysis) and reports are generated as BMP or PDF files, or immediately printed on paper.

The measurement scenario is defined with two steps. First, a measurement procedure (i.e., for pass-by noise test) as well as display settings (e.g., position, instrument type, graph details, and notification panel) are constructed by Sequencer. The second step involves loading the model and setup files when using DEWESoft's measurement software. After compiling the model, the generated code is executed. Then, sensors must be selected (e.g., pressure sensors, temperature sensors, start/stop speed, temperature limitations, sensors for measuring wheel speeds) and their parameters must be set. The procedure continues with the entering of vehicle data, such as the type of vehicle, construction number, and mass of the vehicle. After entering this data, the measurement procedure is ready to start. During the measurements, the end-user can observe the user interface as presented in Fig. 1 with data and graphs, as defined with Sequencer's model.

Figure 1: Driver perspective using DEWESoft and Sequencer

4. Sequencer: A domain-specific modeling language

Fig. 2 shows the architecture of DEWESoft, where two types of Sequencer end-users are shown. In the first group, there are domain experts who are engineers that deal with the construction of measurement procedures. Their task is to model the measurement procedures in Sequencer. The second group comprises users who are testers using prepared measurement procedures. They do not have the ability to modify the measurement procedure. For example, in the automotive measurements, testers are professional drivers. Sequencer leads them through the measurements and helps to avoid any testing errors.

The domain framework is in charge of the entire process of the measurement, as it controls DEWESoft and supervises the hardware. The domain framework is connected with the execution model, which represents an intermediate step between the generation and subsequent execution of the program. The execution model is an internal representation of the measurement procedure. The measurement procedure may consist of three different model types. The first type is a "main model" and is mandatory in the DSML

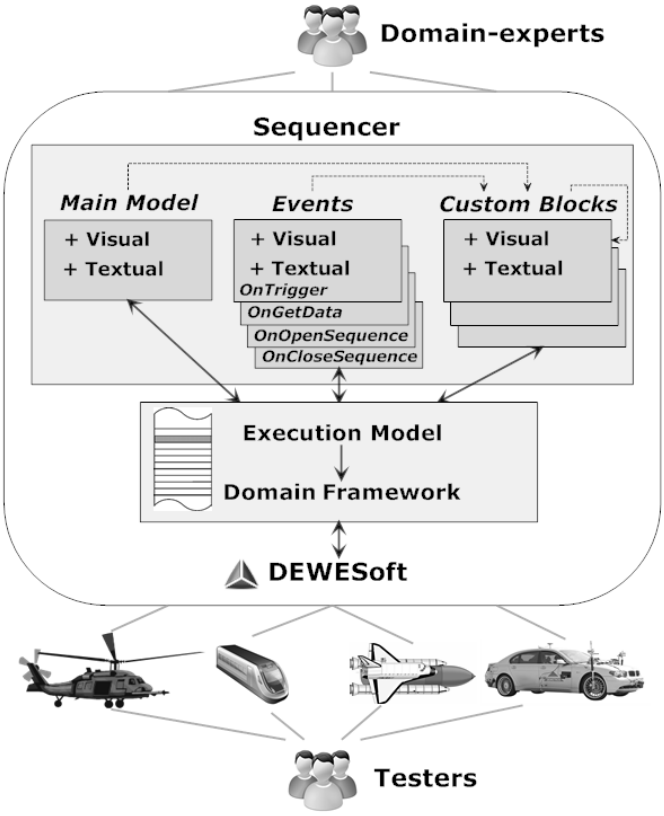

Figure 2: DEWESoft architecture

for Sequencer, because it starts the measurement procedure. Events (e.g., `OnTrigger`) are the second model type. The use of events is optional in Sequencer. Custom models are the third model type. These models can be called "sub-models" and are called by other models or a model of the same type (custom model). The essence of a custom model is in dividing a model into two or more sub-models until these become simple enough to be understood by the domain experts.

The domain expert can choose between two different forms defining a measurement procedure: visual and textual. In both notations, the domain expert saves the program to an XML file. Reusability is thereby gained, as well as the portability of measurement procedures among Sequencer's users. Currently, our debugging support is available only for the visual notation of Sequencer.

4.1. Domain concepts

Usually, DSMLs are developed using metamodels that define the abstract syntax of a DSML. The metamodel includes domain concepts that are used to model a concrete problem. Further explanation of Sequencer's metamodel, as well as domain concepts, can be found in [14]. In this paper, those classes from the metamodel are listed in Table 1, together with the descriptions needed to understand the details of the Ladybird debugger implementation.

In Sequencer, there is a need to start external events at a particular time (e.g., when the brake is pressed, certain parameters need to be calculated). Events interrupt execution of the measurement procedure. As depicted in Fig. 2, each sequence can have four event models (`OnOpenSequence`, `OnTrigger`, `OnGetData`, and `OnCloseSequence`). Events are modeled independently from the main model, but models (even as well as the main model) share the same concepts (see Table 1).

In addition to concept definition, it is necessary to establish the notation for the new language. Visual constructs (also called building blocks) were assigned to classes, which are generally divided into shapes and connections. Each shape belongs to exactly one building block. Each building block represents an action to be executed. The execution of actions begins in the starting building block (marked with a circle) and continues with the connected building block. More information about Sequencer's concepts can be found in [14].

4.2. Execution model

The domain framework serves for communication between the hardware and the rest of the DEWESoft product, and is also responsible for running the execution model. As described earlier, different models (i.e., main model, events and custom models) represent the measurement procedure in Sequencer. Those models are transformed into an execution model that is a unique representation of the models and can be translated back to the models at any time in any notation (e.g., textual, visual, XML). In our case, each notation has its own transformation to an execution model, and vice versa. But, general principles of the transformations are the same. Each language construct (e.g., action, wait, delay) is transformed to an execution block implemented in Object Pascal [44] (DEWESoft is developed in Delphi) as a class with three main functions: `Start()`, `Stop()`, and `Execute()`. As may be inferred, the first function starts the execution, the second function

Table 1: Sequencers' classes

Class	Description
`TAction`	Action is the basic concept of Sequencer. It can initialize hardware, start and stop an acquisition, export measured data, print reports, set different mathematical operations and start many other settings offered by the DEWESoft product.
`TFileManager`	The purpose of the "File manager" is to copy, delete, and rename files, or to launch executable files with special parameters. Later, for instance, it can be useful at the end of a measurement to launch an external program to store measurement data in a specific format.
`TWait`	The "wait statement" is defined with a condition and postpones further execution of Sequencer. An example would be if we wanted to "wait with the measurement until the car enters the measurement zone."
`TDelay`	A "delay" causes the measurement to sleep for "n" seconds.
`TDecision`	A conditional statement helps to define the conditions in the measurement procedure. Conditions may relate to the user's decision or a certain expression. Usually, the value of a certain hardware device (sensor) is tested.
`TRepetition`	A "repetition statement" provides "n" repetitions of certain parts of the model.
`TCalculation`	The calculation is designed to calculate and store values in local and global variables. It is defined with an embedded DSL language. When the formula is calculated, the variable value changes.
`TAudioVideo`	This feature provides audio or video content that can help during the test. The tool supports audio files (*.wav, *.mp3), video files (*.avi, *.mpg) or even an image slideshow presentation (*.ppt).
`TMacro`	This macro can be used to playback the recorded end-user interactions (mouse, keyboard, touch screen).
`TRemoteControl`	With this concept, remote devices (e.g., function generator, shaker) can be controlled via a UART or Ethernet interface.
`TForm`	The purpose of the "Form" concept is to create and display custom windows forms.
`TCustomBlock`	This concept embeds functions, submodels, and subsequences in the current model.
`TConnection`	The concept "connects" different Sequencer concepts.

```
1  function TExecuteBlockW.Execute() : TExecuteStatus;
2  var
3    CondResult : Boolean;
4  begin
5    Result := inherited Execute;
6    CondResult := True;
7    for I := 0 to Conditions.Count - 1 do
8    begin
9      TmpResult := Conditions[I].Decision;
10     if I = 0 then
11       CondResult := TmpResult
12     else if Conditions[I -1].LogOper = loAnd then
13       CondResult := CondResult and TmpResult
14     else if Conditions[I -1].LogOper = loOr then
15       CondResult := CondResult or TmpResult;
16   end;
17   Result := IfThen(CondResult, es_Fineshed, es_Continued);
18 end;
```

Figure 3: `Execute()` function for execution block "Wait for value"

stops the execution, whereas the third function contains the main functionality of the execution block (and hence functionality of a particular language construct). To ensure a good response and synchronization between different parts of the DEWESoft product (hardware and software), the execution block is processed for only a limited period (5 time slices). If a certain action is not completed within a limited period then it is suspended and goes into "sleep" mode (for 10 time slices). It continues to execute after that. When the whole execution block is finished, the `Execute()` function returns status "`es_Finished`". An example of transforming the construct "Wait for value" is given in Fig. 3, while an example for the construct "Delay" is given in Fig. 4.

The automatically generated code in the `Execute()` function for the execution block "Wait for value" loops through logical expressions (lines 7-16, variable `Conditions`). The result of the first logical expression is assigned to the entire condition result (lines 10-11, variable `CondResult`). All other logical expressions are first compared with the logical operators `loAnd` and `loOr` and then combined with previous logical expressions (lines 12-15) for the condition result. If the result of the condition is true, the function returns "`es_Finished`", otherwise it returns "`es_Continued`".

Figure 4 shows an example for function `Execute()` for execution block "Delay". First, it checks if the delay is greater than one execution slice (line 3). If the result is true, the function waits for the whole execution slice (line 5). The `Sleep` function takes an integer number; therefore, function

```
1  function TExecuteBlockD.Execute() : TExecuteStatus;
2  begin
3    if FDelay > MAX_SLICE_TIME_S then
4    begin
5      Sleep(Trunc(MAX_SLICE_TIME_S * 1000));
6      FDelay := FDelay - MAX_SLICE_TIME_S;
7      Result := es_Continued;
8    end
9    else
10   begin
11     Sleep(Trunc(FDelay * 1000));
12     FDelay := 0;
13     Result := es_Finished;
14   end;
15 end;
```

Figure 4: `Execute()` function for execution block "Delay"

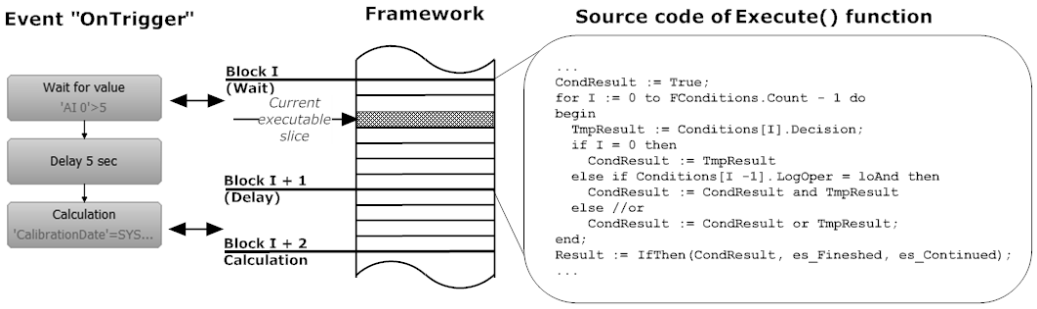

Figure 5: Transformation of Sequencer constructs into execution model

`Trunc` is used to cut off the decimal part of a number. Also, the number is multiplied by 1000 to construct the time in milliseconds. In this case, the function returns status "`es_Continued`". Otherwise, if the delay is less than one execution slice (lines 10-14), it waits for the delay time and returns status "`es_Finished`".

Transformation between Sequencer's constructs (e.g., a building block in the visual notation or sentence in the textual notation) and execution blocks is one-to-one, as shown in Fig. 5. On the left side of Fig. 5, part of the event model (Event "`OnTrigger`") is visible. It consists of three building blocks: the first block checks the condition (value of variable "AI 0" must be greater than 5 - "AI 0" is the first input analog channel, which stands for acceleration channel), the second block delays execution for 5 seconds, and the third building block performs the required calculations. Those building blocks are transformed into three execution blocks (see also Figs. 3 and 4).

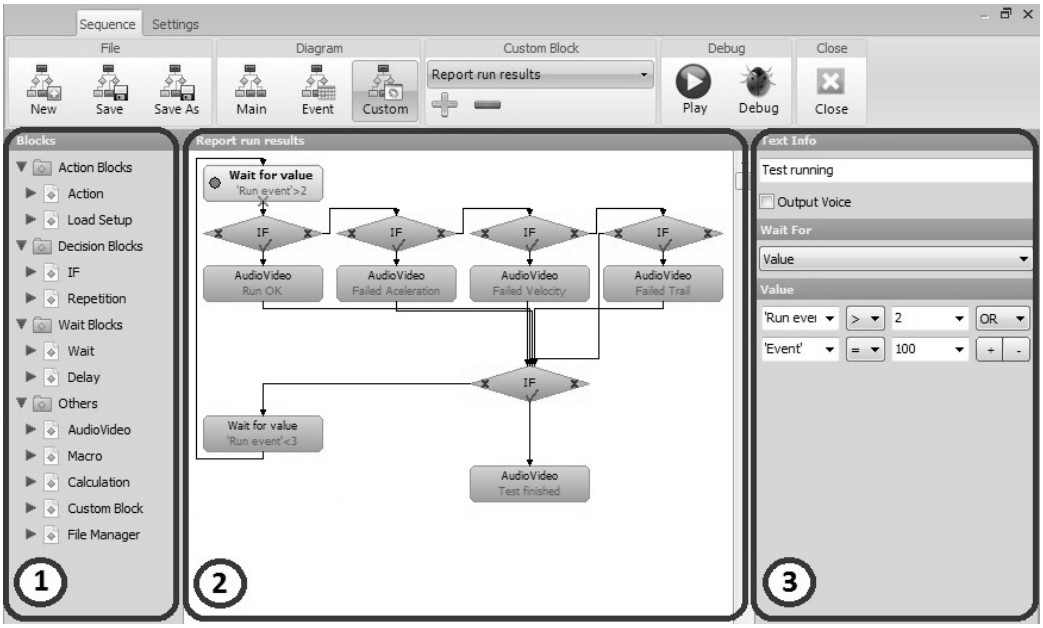

Figure 6: Sequencer's modeling environment

4.3. Modeling with Sequencer

Fig. 6 shows the modeling tool, where the main program and events can be modeled by domain experts. In the modeling tool, building blocks (e.g., action, wait) are visible on the left side of the user-interface (part one). Building blocks can be used in the modeling area (part two) using drag and drop functionality. In part three, the default values of global and local variables can be set. More specifically, the model in Fig. 6 shows a custom model where a report for a test-driver is modeled. The test-driver will be reported with different audio-video content depending on different results obtained by data acquisition. At the beginning, the sequence waits for a condition "Wait for value: 'Run event'> 2" (after a certain number of event runs, the information is presented on the end-user's display). After that, the measurement procedure checks the run status and notifies the testers (e.g., AudioVideo: "Failed Velocity").

Additionally, a domain expert can define the static settings (e.g., component positions on an end-user's display) of DEWESoft in the modeling tool. Also, custom forms for users can be designed at this level. The form constructs (e.g., label, text box, combo box) can be linked to the acquired values

Table 2: Problem scenarios suggesting the need for improved debugging support

Problem	Description
P1a	During execution, the sequence can fall into an infinite loop. This problem means that the sequence cannot be stopped. Also, it is impossible to skip the current building block and proceed with another. Moreover, it is impossible to pause/resume the execution of the sequence.
P1b	When domain experts run very complex sequences, it is difficult to find bugs, because bugs can occur in the model or in the measuring equipment. It would be helpful if we could execute the model step-by-step (i.e., one building block at a time) which would improve control over the execution.
P2	For improved understanding of the sequence, the domain experts would prefer some kind of animation at sequence runtime.
P3	In some cases, domain experts would like to stop the execution before or after a specific building block in the model. In some cases, they would also like to stop the sequence at a specific trigger (for example, when the brake pedal is pressed).
P4	Some applications are time critical. One example of such an application is testing the water heater, wherein the measurement lasts for a long time.
P5	For faster testing, domain experts often want to have the ability to monitor and alter the values of a running sequence.

from the measurement. Domain experts can use math modules, as well as subsequent plotting graphs and drafting reports, which can be presented to users during their execution.

5. Ladybird - debugging support in Sequencer

In Table 2, some limitations from user studies observed by the domain experts were reported before debugging support was incorporated in Sequencer. This information was obtained through customer support service and seminars that DEWESoft organizes every year (one session is also about Sequencer). The problems regarding realistic applications and requests have been divided into six different categories.

Most of Ladybird's features to solve problem scenarios from Table 2 can be found in Fig. 7, which shows Sequencer during a debugging process. The

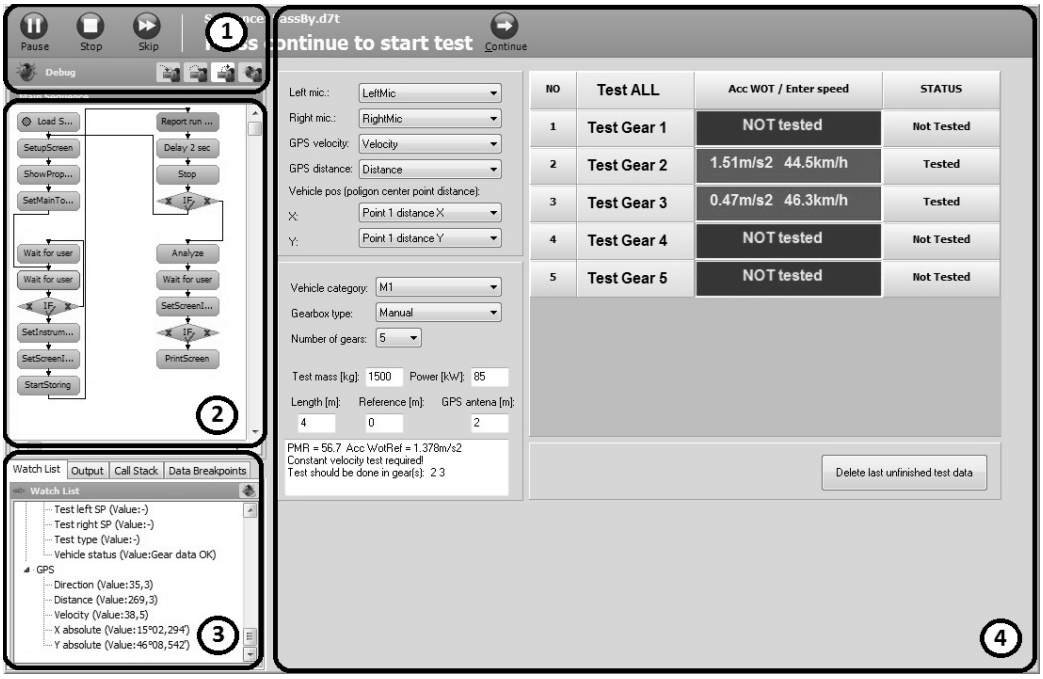

Figure 7: Debugging mode in Sequencer using Ladybird

first, second, and third parts on Fig. 7 contain various Ladybird features (e.g., controls, steps, animation, watch list, output view, call stack, and data breakpoints), while on the right side of the screen (part 4), the domain expert can observe the execution of the measurement system. More specifically, in part 4 of Fig. 7, the execution of a "pass-by" noise test is presented. On the left side, optional parameters of a test can be set, while the right side shows information of test competition for a specific gear.

In the next subsections we describe functionalities of the Ladybird debugger (available debugging comments and actions), as well as their intended effects. All of the problems mentioned in Table 2 have been addressed by new functionalities offered by the Ladybird debugger. We provide examples with a "pass-by" noise test.

5.1. Execution Modes and Steps

A sequence can be executed in release or debug mode. In release mode, the domain expert can only play, suspend or stop the sequence. To be able to use different debugging features, the sequence must be run in debug mode.

In this mode, an end-user has the ability to execute the model continuously or step-by-step (see first part of Fig. 7).

Controls

In order to solve problem P1a, we added the controls "Stop", "Pause/Play", and "Skip". The first control allows the domain expert to terminate the sequence. The second one halts the execution of the sequence. When the sequence execution is halted, the domain expert can continue execution with the "Play" control. The "Skip" control allows skipping the current execution block. In addition, "Skip" can control execution, so that some execution block waiting for a certain action is simply skipped. For example, in the pass-by noise test sampling, noise emission values start from one point to another. It might happen that from unknown reasons sampling data did not stop at a specific polygon point. With this control, the execution block responsible for acquiring data can be skipped.

Steps

This feature contains three types of steps: "Step Over", "Step Into", and "Step Out", which help to solve problem P1b (also in the first part of Fig. 7). The first control enables the execution of a building block at once, skipping the sub-model. The second control allows changing the current scope and enables stepping into a sub-model (custom model). If the "Step Into" is selected and the user wants to exit from the current context to the parent model, the "Step-Out" control can be used. This control tells the debugger to run until the end of the current model (scope) and return one-level higher. For example, if the pass-by noise test hypothetically has some strange behavior (e.g., when changing the speed or gear does not affect acquired values of noise emission) and we do not know the cause, we would step through the program. We observe the acquired values and local variables in Sequencer for every step and try to find the problem.

5.2. Animation

Sometimes non-experienced users do not comprehend the meaning of the returned values by Sequencer. Although they are domain experts and have domain knowledge, they may not understand the returned values. Therefore, a visual presentation or animation of the current model state can be a useful aid in order to make the perception easier and clearer.

Even in our DSML, similar problems occurred, so that a new feature in the form of an animation was developed that solved problem P2. Another purpose of this feature was to help the domain expert inspect the control flows of a given model and to understand its behavior. Visualization consists of a static and dynamic part. The first is a model presentation like in the modeling tool, whereas the dynamic part is an animation on that model at runtime (see part two in Fig. 7). For example, during the pass-by noise test the end user can observe how the measurement procedure is executed. The user can see the crossing of different polygon points (e.g., driving past the microphones) and detect the current location in the maneuver.

5.3. Breakpoints

Very complex and complicated sequences are almost impossible to follow using step-by-step features (e.g., problem scenario P3). Therefore, it is necessary to use other mechanisms. From the GPL languages, this feature is known as a "breakpoint", which temporarily suspends the execution of a program at a certain statement in the code. In Sequencer, the same technique was used where the breakpoint can be applied to building blocks in the modeling area of the tool. Domain experts have the ability to specify when or where the breakpoint should interrupt a normal sequence execution. The suspension can be considered and determined in two ways: the source or data breakpoints.

Source breakpoint

The source breakpoint is attached to the building block contained by the model. The feature also supports associating "pass count", which enables domain experts to specify an arbitrary number that determines how many times the breakpoint can be executed before the debugger stops the execution. Moreover, this breakpoint feature also supports a conditional expression that determines whether the execution should be halted. This is useful if we want to halt the execution only in the case when the signal from the measurement returns a certain value. For example, we would like to stop the program when a tester enters parameters for the pass-by noise maneuver. After that we would check if the entered values are assigned correctly in the variables. In that case, we would add a source breakpoint in the model at the position where the values are entered. At the stop position in the program, the variables would be checked.

Data breakpoint

The second type is a data breakpoint, where the execution stops when the value of a local or global variable is changed. For example, in the pass-by noise test, the test should be made in the 2^{nd} gear. A sequence can be stopped automatically by a breakpoint when the test-driver makes a change from one gear to another.

5.4. Print statements

In some cases, debugging in the form of steps and breakpoints is difficult. It is thus necessary to find other solutions that have no direct impact on the execution itself. To this end, we use a known GPL technique: "print statements", which are generally used to output local and global variables and to verify that building blocks are executed at certain points of execution. This information can be observed in the output window and can be stored directly to the text file. After a long test, the stored values can be studied by domain experts. For example, during a pass-by noise test we can check noise emission of different vehicle parts. In such cases, debugging in the form of steps and breakpoints and following all emission values is difficult (problem scenario P4).

5.5. Variable View and Stack Traces

In order to solve problem scenario P5, Sequencer provides an option for changing values of local and global data variables (also called channels) at runtime. We can monitor various measured signals (e.g., analog, digital, counter) and the signals that come from different data buses (e.g., CAN, FlexRay, ARINC 429 and MIL-STD-1553). For example, during debugging of the pass-by noise test the user can observe local variables from Sequencer and specific signals that are not present in the end-user GUI (because it is not important for a measurement test), but could be useful to locate the bug in the measurement. In addition, this feature allows one to see the model level of the currently executed building block. The domain expert can see which of Sequencer's constructs have led the execution into its current state. The feature is called "stack traces" and becomes visible when the execution is suspended.

6. Ladybird's implementation details

DSLs can be divided into three categories: imperative, declarative and hybrid [29]. An imperative DSL is centered on assignment expressions and

control flow statements, which allow for the changing of the content of cells in memory at runtime. A declarative DSL is based on stating the relationship between inputs and outputs. A hybrid DSL is a mixture of the first two. Each DSL category requires different debugging approaches (e.g., algorithmic debuggers, declarative debuggers, event-based debuggers, assertion checkers, and debugging queries). A DSML debugger is situated in a similar position as a DSL debugger - instead of checking programs, a debugger is used to debug models. In Sequencer, we solely focus on imperative DSML debuggers. Our debugging tool (Ladybird) can be used to find modeling bugs at runtime in a visual manner.

6.1. Modifications of the execution model

In order to implement a debugger for Sequencer, its execution model was slightly changed. The Ladybird architecture, as illustrated in Fig. 8, shows how we designed the debugger to achieve our goals. In general, the Ladybird debugging architecture has three levels. The DEWESoft measurement system and Sequencer are in the first level. In this level, we changed the execution model so that it calls interfaces from the communication components (on the second level), which are responsible for building internal structures of significant information needed by the debugger. In this way, those components facilitate interaction between Sequencer's execution model and the Ladybird debugger (on the third level). For the latter, the following features were implemented: execution modes, steps, breakpoints, print statements, model animations, variable views, and stack traces.

6.2. Communication components

Ladybird internally communicates with Sequencer via four communication components:

1. Source Model Mapping
2. Debugging Actions
3. Execution Response, and
4. Memory Mapping.

The Source Model Mapping Component (SMMC) is a binding process that indicates the location of the target execution framework code that corresponds to a single building block in the source DSML. The main purpose is to map back from the execution block into Sequencer's construct. The mapping components store the information in an internal representation that

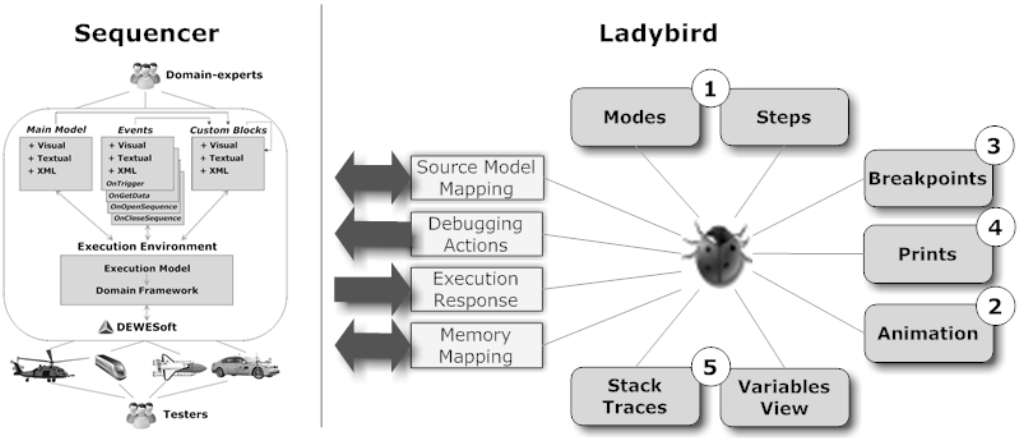

Figure 8: Overview of Ladybird's architecture

contains the model type, group model number, building block number and current time slice in the framework. The model type can be the main model, custom model and event model. The second element is a group model number and indicates the unique number for the custom and event model. The main model is always unique; therefore, the group model number is always "0". The group model number for an event model represents `OnGetData` (unique value "0"), `OnTrigger` (unique value "1"), `OnOpenSequence` (unique value "2") or `OnCloseSequence` (unique value "3"). Custom models receive a group model number with regard to the sequence of their occurrence (the first custom block receives unique value "0" and last custom model "n-1"). The third element is a building block number that is a unique value for the building block from the model. The last element presents the slice number, which is the current execution's position in the framework, where the programming pointer is indicated. Note that execution blocks are processed in time slices (maximum of five time slices), then they go into sleep mode (maximum of ten time slices), when the domain framework executes other operations (e.g., data acquisition). If a certain execution block is not completed within a limited number of slices, it is suspended and continues to execute later.

In such a manner, using the SMMC component, it is always possible to map an execution block back to a Sequencer construct. Fig. 9 shows an example of how to map the current execution block in the framework to the

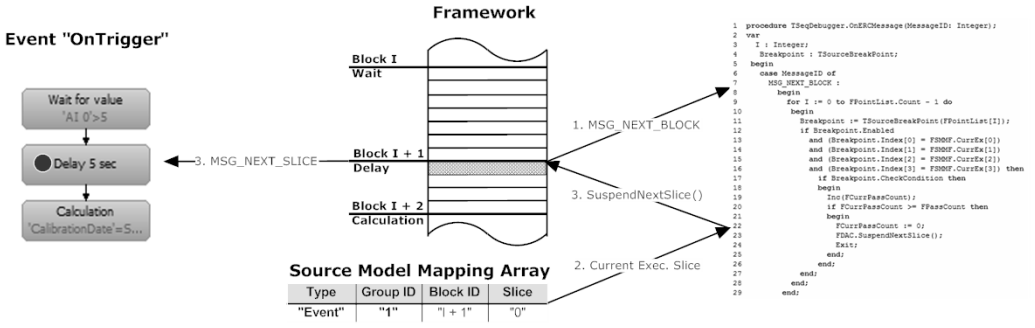

Figure 9: Source Model Mapping in the Ladybird tool

DSML construct, and shows the same event model as Fig. 5. The model mapping array of a current building block consists of the following values: Event type, "1" as unique Group ID, "I+1" as current block and the slice number "0".

The Debugging Actions Component (DAC) consists of an interface between the domain framework and the Ladybird debugger. This mechanism ensures the termination, suspending, skipping or resumption of the current executable model. The DAC interface has four commands that impact the execution of the program: `SuspendNextSlice()`, `ResumeCurrentSlice()`, `GoToNextBlock()` and `TerminateProgram()`. The first command performs the current slice and suspends the execution of the next slice if it exists. The opposite of the first command is the `ResumeCurrentSlice()` command, which continues the execution. The third command performs the current slice and then jumps to the next execution block in the queue. The command `TerminateProgram()` stops the execution of the entire program.

The Execution Response Component (ERC) ensures that the debugging tool knows when and how the execution of the current program was performed. In general, the domain framework informs the debugging tool to send special messages such as `MSG_NEXT_SLICE` and `MSG_NEXT_BLOCK`. The first and second messages indicate that the previous slice/block has been performed and that the next slice/block is ready to be performed.

In addition to the previously described components, the debugging tool also needs to have access to the memory, which requires a Memory Mapping Component (MMC). It allows the observation of changes to global or local variables at runtime. Moreover, the MMC interface has been constructed in

such a way that the Ladybird debugger can get/set all the information of the acquired signal (e.g., signal type, sample rate, resolution), which helps to construct a precise debugging tool.

6.3. Debugger implementation

In this subsection, the individual implementation of several debugging features is explained. The communication components (Section 6.2) serve to control the execution of the specific debugging features. This is achieved using message passing between specific debugging features and communication components.

6.3.1. Execution Modes and Steps

Controls

The fundamental functionalities in our debugging tool represent controls such as Play/Pause, Stop and Skip. These functionalities help us to have impact over the program execution. The algorithm for those functions is simple and all functions require communication with the DAC component. The playing and pausing commands simply call functions `ResumeNextSlice()` and `SuspendNextSlice()`, which suspend/resume the measurement procedure. Stopping a command presents immediate termination of the program and the function `TerminateProgram()` should be called. The Skip command uses the `GoToNextBlock()` function.

Steps

Step algorithms in the Ladybird tool can be performed only if the execution is suspended. These algorithms first send the `ResumeCurrentSlice()` command in the DAC and wait until the debugging tool gets the message `MSG_NEXT_BLOCK`, which means that the next building block is ready to be performed (in Fig. 5, first block called "Wait for value" would be resumed and the next block called "Delay" would be ready to be performed). After that message, the tool checks the condition that is different for each stepping type (e.g., "Step Over" has to check if the building block is not in the new sub-model via the SMMC mechanism). If the above condition passes, the algorithm sends the command `SuspendNextSlice()` (in our example, the second block would be suspended) and waits for the next end-user interaction. Otherwise, the tool waits for the next ERC message. The sequence diagram for the "Step Over" interaction is shown in Fig. 10.

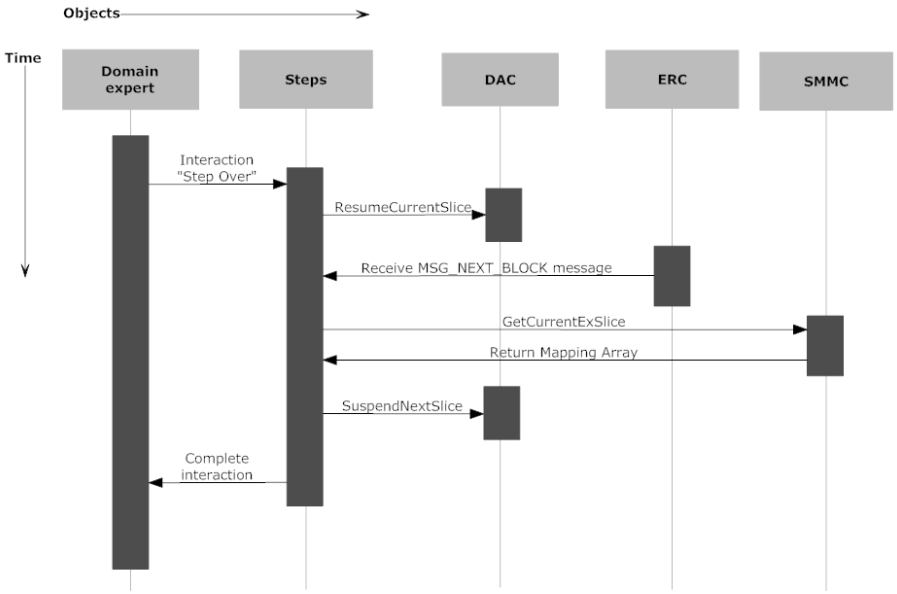

Figure 10: Sequence diagram for "Step Over" interaction

6.3.2. Animation

Fig. 11 shows how the animation is handled by the ERC messages. On those messages, the animation refreshes the status and repaints the preview panel. In our case we are using `MSG_NEXT_SLICE`, which more precisely indicates an animation that the domain expert can observe so that the model's behavior is clear. Moreover, the animation uses an SMMC component, which helps to determine the current execution block in the framework.

6.3.3. Breakpoints

Fig. 12 shows part of the implementation of the source breakpoint, which is written in the Delphi programming language [44]. The written function is invoked when it receives the message (line 7) that can check whether it is necessary to stop the implementation of the sequence. This message is obtained only once before the first slice would be performed and for that reason, the execution block cannot be interrupted during the execution. First, we go through each breakpoint (line 9) and check if the current source point corresponds to a specific breakpoint (lines 12-17), where `FSMMF.CurrEx[0]` represents a source point's model type, `FSMMF.CurrEx[1]` represents the group model number, etc. (see Fig. 9, again). If the condition is a success, we increase the current pass count (line 19) and check the new pass count (20).

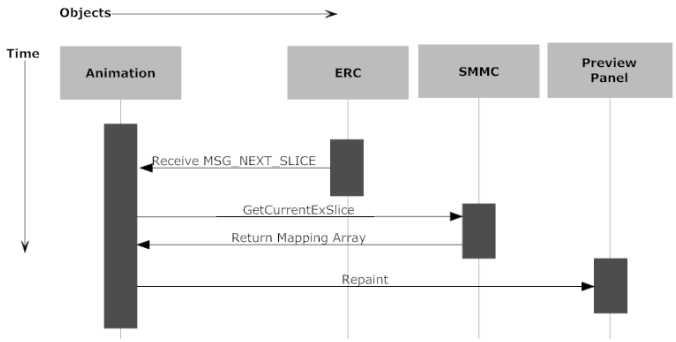

Figure 11: Sequence diagram for animation

```
1  procedure TSeqDebugger.OnERCMessage(MessageID: Integer);
2  var
3    I : Integer;
4     Breakpoint : TSourceBreakPoint;
5   begin
6     case MessageID of
7       MSG_NEXT_BLOCK :
8         begin
9           for I := 0 to FPointList.Count - 1 do
10           begin
11             Breakpoint := TSourceBreakPoint(FPointList[I]);
12             if Breakpoint.Enabled
13               and (Breakpoint.Index[0] = FSMMF.CurrEx[0])
14               and (Breakpoint.Index[1] = FSMMF.CurrEx[1])
15               and (Breakpoint.Index[2] = FSMMF.CurrEx[2])
16               and (Breakpoint.Index[3] = FSMMF.CurrEx[3]) then
17                 if Breakpoint.CheckCondition then
18                 begin
19                   Inc(FCurrPassCount);
20                   if FCurrPassCount >= FPassCount then
21                   begin
22                     FCurrPassCount := 0;
23                     FDAC.SuspendNextSlice();
24                     Exit;
25                   end;
26                 end;
27             end;
28           end;
29         end;
```

Figure 12: Part of the source breakpoints implementation

After that we call a function (line 23) to stop the framework execution process.

6.3.4. Print statements

This algorithm works exactly like the algorithm for source breakpoints. The difference between the algorithms is only in the fact that, in the Print algorithm, the condition does not suspend the execution, but logs output val-

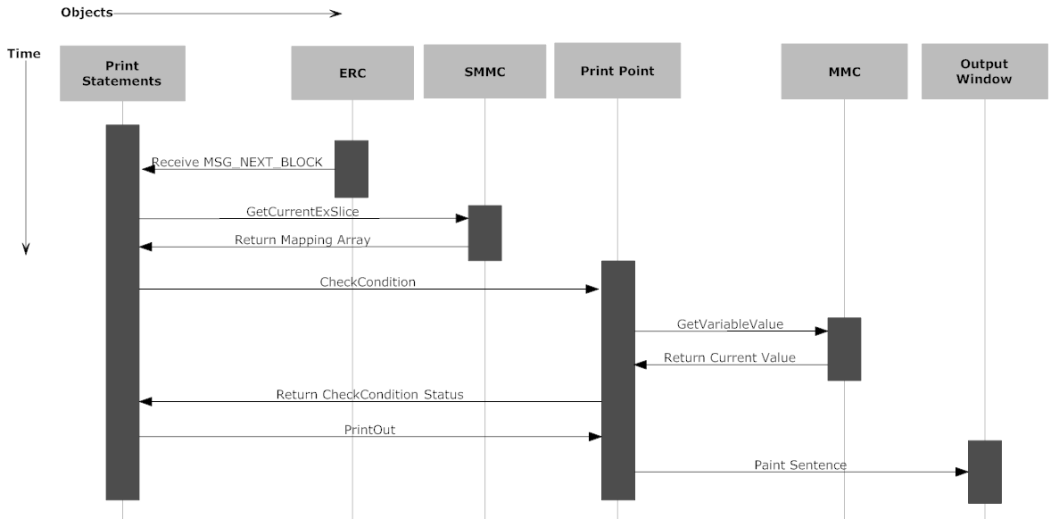

Figure 13: Sequence diagram for printing trace information

ues. Fig. 13 shows the sequence diagram for the print statement algorithm. First, this algorithm waits to receive ERC message `MSG_NEXT _BLOCK`. On that message, the algorithm checks if the current source point corresponds to a specific print statement through the SMMC component. After that, a condition (e.g., expression, pass count, and expected results) needs to be checked. If the condition status is passed, the print statement is displayed on the screen or stored into the text file.

6.3.5. Variable View and Stack Traces

Fig. 14 shows the sequence diagram demonstrating how a variable can be changed by a domain expert in the execution model. During refresh time, when the variables are updated, the Variable View requires all global and local variables (`GetAllVariables()`) via the MMC component. The current acquired values from memory can be shown in the Variable View where the domain expert can change the value of the variable. This interaction calls function `SetVariableValue()`, which sets the new value in memory. Moreover, at the same time the stack traces allow the domain expert to see the current position in the model. This algorithm requires the execution path (stack) from the SMMC.

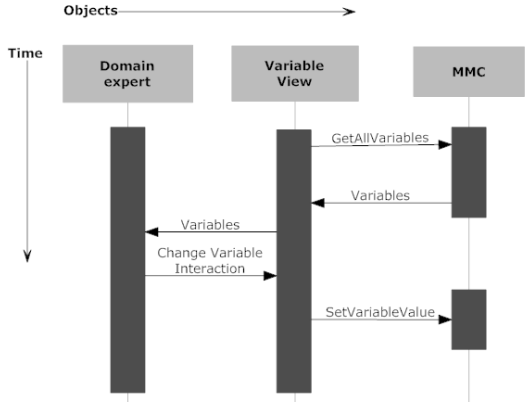

Figure 14: Sequence diagram for Variable View

7. A demonstration of Ladybird

The debugging tools in traditional software development environments can assist programmers and help to make them more productive [45]. We have found the same to be true with the Ladybird debugging features. Ladybird can increase the quality of models by helping end-users find and correct errors. Those errors that occur during measurement can now be minimized during debugging. One concrete example where Ladybird was used to find and correct an error is given further in this section.

Even before Sequencer, DEWESoft's users could add new functionality independently (via a plug-in) and control DEWESoft itself through the DCOM interface. However, comparing a DCOM application of pass-by noise test with a program in Sequencer in Table 3 shows that the productivity of end-users increased rapidly when using Sequencer. Our observations have shown that end-users can develop applications faster, while the program length has been reduced in some cases by a factor of 20.

We noticed that after release of DEWESoft 7.0, when Sequencer was incorporated in the measurement software, the number of errors decreased [17]. However, there were situations where end-users still had problems with detecting errors and finding the solutions. Therefore, we decided to implement the Ladybird debugger, which works at the level of the domain, to improve end-user comprehension of information received through debugging facilities. One typical example where end-users had a problem discovering an error was the following: when some signal returned by the measurement system had a different value than expected, users could not distinguish whether this value

Table 3: Comparison of Sequencer application with DCOM applications in LOC

Measurement procedures	**DCOM Application**	**Sequencer program**
Pass-by noise test	541	30
Double lane-change test	329	16
Functional Safety test	347	19
Lane departure test	336	17

was connected with a failure on the measured object, or with an improper measurement equipment setting. Such a problem was solved with the use of Ladybird. In the real case scenario, we first set the hardware and software for the pass-by noise test, and then we tested the whole measurement system. During the initial test, we found that some optical signal had the same values during measurement. To discover the problem, we started debugging when the car started moving and then in a step-by-step manner we executed the testing procedure. However, step-by-step execution was not convenient when we wanted to observe the speed value. Therefore, we changed the debug mode from step-by-step execution to (usual) debug mode execution. In this case, the current blocks and the values of variables were stored in a text file.

In our case, we applied the following procedure for the detection of the bug. We used the data breakpoint where we set the condition that the sequence will stop when the value of the variable "velocity" was higher than 10 km/h. After that, we executed a step-by-step sequence that monitored variables until the velocity was higher than the entered speed, when the starting line optical signal was triggered on the raising edge. Then, we added the print block and set it to store the result of the values of the current variables. During debugging, we discovered that the optical sensor was not plugged in correctly, because the values were the same even if we interrupted the measurement of the vehicle at the start line. Fig. 7 shows the debugging mode in Sequencer for a measurement procedure for the pass-by noise test. The first part of the screenshot marks the step controls. The second part in Fig. 7 shows the animation where the domain-expert can observe the progression of the model execution. The watch list, output view, call stack,

and data breakpoints are shown in the third part. On the right side of the screen, the domain-expert can observe the execution and result of the measurement system (part four).

8. Discussion

In this section, the experience of developing a DSML and its debugger are introduced. Firstly, several details of Sequencer DSML are discussed. Then, Ladybird's development effort is summarized. At the end of this section, we discuss some ideas on how to generalize the debugging features from Ladybird so that the implementation can be reused for other DSML debuggers.

8.1. Developing a DSML with conventional techniques

Although the use of MDE tools (e.g., MetaEdit+, GME, EMF) has been shown to be effective in many cases [46], [27], in our case, these tools could not be applied for the following reasons. First, the integration of Sequencer DSML with DEWESoft (a massive product containing over 900,000 LOC, and developed in more than 50 engineer years) was challenging. One of the requirements was that software remains in one form – external use of a metamodeling environment was not an option. Therefore, connecting DEWESoft's base language (Delphi), with the language of a metamodeling environment (provided by a metamodeling tool), would be a demanding task. A second reason is the language of the generated code by metamodeling tools – only those metamodeling tools would be appropriate that support generation of Delphi code (if we eliminate source-to-source transformation and similar solutions). Most of the MDE tools mentioned previously offer support for DSML development, but fail to provide assistance for DSML debugging.

Currently, Sequencer contains over 24,000 LOC, which provides implementation support for the debugger and other features of Sequencer DSML. Developing a DSML with a conventional technique required additional implementation effort, because we have implemented the object-oriented structure that represents classes in the metamodel. However, this represents just 5.9% of Sequencer's code, because the semantics (transformations) needed to be written in both cases. Of course, with metamodeling tools the modeling tool is generated automatically, but in our case we had to implement the editor manually - this represented 24.2% of Sequencer's code. Overall, the conventional technique for implementation of a DSML required approximately 7,000 LOC which could be generated when using MDE tools. However, we

believe that the implementation of the runtime execution environment allowed us to implement the debugger facilities more easily. Because current metamodeling tools do not support the development of debuggers at such a modeling level, the implementation of the debugger inside or outside of a metamodeling tool can be a very challenging task.

8.2. Debugger development effort

The Ladybird debugger was implemented after Sequencer was deployed to market. The customers' need for better understanding of data acquisition results led the DEWESoft team to the decision to develop the debugging support. Ladybird's implementation was done in three engineering months. Since the first release, updates were introduced, which are not counted towards the development time. Ladybird contains over 2.5 kLOC, which represents almost 10.5% of Sequencer. The Ladybird debugger was developed by the same engineers who designed the Sequencer DSML. Those engineers had no previous experience in implementing a debugger for a DSML. These facts certainly have influenced the development time of these tools. The time of development and the debugger's size in LOC must be considered with caution. Other domains might differ substantially, reflecting the different needs of the domains, customers and underlying framework, resulting in quite possibly a different implementation effort.

8.3. Generalization of debugging features

In general, debugging support was developed to help experts detect errors in their measurement procedures. Because the debugger is implemented separately from the underlying DSML, most of the concepts implemented (execution modes, steps, controls, brakpoints, etc.) can also be generalized to other imperative languages (except for animation, where domain-specific constructs are animated). In that case, some minor changes have to be introduced to Ladybird. Language engineers should partially re-implement Ladybird's communication components: DAC, ERC, MMC and SMMC, that link a language with its debugger. For instance, in DAC functions `SuspendNextSlice()`, `ResumeCurrentSlice()`, `GoToNextBlock()`, and `TerminateProgram()`, the model execution should be re-implemented for the new language domain. Component ERC should return correct messages when the individual block or slice is performed. In SMMC, the language engineer would have to change the linking table (that connects DSML constructs and execution blocks). In MMC, methods Get and Set should be implemented

again to return values of local and global variables. Note that actions and message sequences (function calls) inside communication components remain almost unchanged. This high-level logic can be reused completely in the debbuger implementation of another DSML. However, those assumptions need a deeper analysis and further studies to confirm our beliefs, which could be a subject of our future research studies.

9. Conclusion

Although the domain experts we observed were able to create a measurement procedure quickly and easily in Sequencer, they encountered several problems (see Table 2). To support their development in Sequencer and further improve user productivity, additional tool support was necessary. This paper presents the debugging features in Sequencer. The modeling environment offers control features, such as model step-wise execution, breakpoint functionality and model behavior animation. With the Ladybird debugger, it is also possible to simultaneously watch multiple models and, during their operation, monitor and alter local and global variables. From our observation, debugging tools are essential for a DSML to be successful. New tools (e.g., debuggers) that are common in traditional IDEs need to be more present in DSML modeling environments.

10. Acknowledgment

This work was partly sponsored by the European Social Foundation. Moreover, we would like to thank DEWESoft co-workers who contributed to a discussion of DSML problems. This work was also sponsored in part by NSF CAREER grant CCF-1052616.

References

[1] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM SIGPLAN Notices 35 (6) (2000) 26–36.

[2] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.

[3] M. Fowler, Domain-Specific Languages, Addison Wesley, 2011.

[4] T. Kosar, N. Oliveira, M. Mernik, M. J. Varanda Pereira, M. Črepinšek, D. da Cruz, P. R. Henriques, Comparing general-purpose and domain-specific languages: An empirical study, Computer Science and Information Systems 7 (2) (2010) 247–264.

[5] T. Kosar, M. Mernik, J. Carver, Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments, Empirical Software Engineering 17 (3) (2012) 276–304.

[6] D. S. Wile, Supporting the DSL spectrum, Journal of Computing and Information Technology 9 (4) (2001) 263–287.

[7] S. Mauw, W. Wiersma, T. Willemse, Language-driven system design, International Journal of Software Engineering and Knowledge Engineering 6 (14) (2004) 625–664.

[8] C. Elliott, An embedded modeling language approach to interactive 3D and multimedia animation, IEEE Transactions on Software Engineering 25 (3) (1999) 291–309.

[9] S. Thibault, R. Marlet, C. Consel, Domain-specific languages: From design to implementation application to video device drivers generation, IEEE Transactions on Software Engineering 25 (3) (1999) 363–377.

[10] S. Peyton Jones, A. Blackwell, M. Burnett, A user-centred approach to functions in Excel, in: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP), 2003, pp. 165–176.

[11] B. Bryant, J. Gray, M. Mernik, Domain-specific software engineering, in: Workshop on the Future of Software Engineering Research (FoSER), 2010, pp. 65–68.

[12] S. Kelly, K. Lyytinen, M. Rossi, MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment, in: Proceedings of the 8th International Conference on Advanced Information Systems Engineering (CAiSE), 1996, pp. 1–21.

[13] J. Sprinkle, M. Mernik, J.-P. Tolvanen, D. Spinellis, Guest editors introduction: What kinds of nails need a domain-specific hammer?, IEEE Software 26 (4) (2009) 15–18.

[14] T. Kos, T. Kosar, M. Mernik, Development of data acquisition systems by using a domain-specific modeling language, Computers in Industry 63 (3) (2012) 181–192.

[15] J. Knez, M. Tuma, G. M. Smith, A new approach to measurements - the PC instrument, Sound and Vibration 36 (5) (2002) 16–20.

[16] Data acquisition software DEWESoft 7, available at `http://www.dewesoft.com/`.

[17] T. Kos, T. Kosar, J. Knez, M. Mernik, From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer, Computer Science and Information Systems 8 (2) (2011) 361–378.

[18] VBOXTools Software, available at `http://www.racelogic.co.uk/`.

[19] Corrsys-datron, available at `http://www.corrsys-datron.com/`.

[20] P. Wadler, Why no one uses functional languages, ACM Sigplan Notices 33 (8) (1998) 23–27.

[21] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, J.-P. Tolvanen, DSLs: The Good, the Bad, and the Ugly, in: Companion to the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications (OOPSLA), 2008, pp. 791–794.

[22] S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, John Wiley & Sons Inc., 2008.

[23] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, The generic modeling environment, in: Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP), 2001.

[24] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd Edition, Addison-Wesley, 2008.

[25] MPS, available at `http://www.jetbrains.com/mps/`.

[26] G. Booch, J. Rumbaugh, I. Jacobson, Unified Modeling Language User Guide, 2nd Edition, Addison-Wesley, 2005.

[27] A. Gokhale, D. Kaul, A. Kogekar, J. Gray, S. Gokhale, POSAML: A visual modeling language for middleware provisioning, Journal of Visual Languages and Computing 18 (4) (2007) 359–377.

[28] J. White, J. Hill, J. Gray, S. Tambe, A. Gokhale, D. Schmidt, Improving domain-specific language reuse with software product line techniques, IEEE Software 26 (4) (2009) 47–53.

[29] H. Wu, J. G. Gray, M. Mernik, Grammar-driven generation of domain-specific language debuggers, Software Practice and Experience 38 (10) (2008) 1073–1103.

[30] H. Wu, J. Gray, M. Mernik, Unit testing for domain-specific languages, in: Proceedings of the International Federation for Information Processing: Theory and Practice 2 (IFIP TC 2) Working Conference on Domain-Specific Languages, 2009, pp. 125–147.

[31] R. Mannadiar, H. Vangheluwe, Debugging in domain-specific modeling, in: Proceedings of the 3rd International Conference on Software Language Engineering (SLE), 2010, pp. 276–285.

[32] J. de Lara, H. Vangheluwe, M. Alfonseca, Metamodeling and graph grammars for multi-paradigm modelling in AToM3, Software and Systems Modeling 3 (3) (2004) 194–209.

[33] M. Hibberd, M. Lawley, K. Raymond, Forensic debugging of model transformations, in: Proceedings of International Conference on Model Driven Engineering Languages and Systems (MODELS), 2007, pp. 589–604.

[34] A. Blunk, J. Fischer, D. A. Sadilek, Modelling a debugger for an imperative voice control language, in: Proceedings of the International Conference on System Design Languages, 2009, pp. 149–164.

[35] Eclipse Foundation: Eclipse Modeling Framework (EMF), available at `http://www.eclipse.org/modeling/emf`.

[36] Eclipse Foundation: Eclipse Debugging Framework, available at `http://help.eclipse.org/ ganymede/index.jsp?topic= /org.eclipse.platform.doc.isv/guide/debug.htm`.

[37] T. Kos, T. Kosar, M. Mernik, J. Knez, Ladybird: Debugging support in the Sequencer, in: Applications of mathematics and computer engineering, WSEAS Press, 2011, pp. 135–139.

[38] C. Elliott, V. Vijayakumar, W. Zink, R. Hansen, National instruments labview: A programming environment for laboratory automation and measurement, Journal of the Association for Laboratory Automation 12 (1) (2007) 17–24.

[39] FlexRay, available at `http://www.flexray.com/`.

[40] 2009 product of the year winners, NASA Tech Briefs 34 (4) (2010) 10.

[41] P. M. Pachlhofer, J. W. Panek, D. J. Dicki, B. R. Piendl, P. J. Lizanich, G. A. Klann, Advances in engine test capabilities at the NASA Glenn Research Center's Propulsion Systems Laboratory, in: Proceedings of the American Society of Mechanical Engineers (ASME), 2006, pp. 101–108.

[42] R. Bosch, Safety, Comfort and Convenience Systems, John Wiley & Sons Inc., 2006.

[43] Automotive Directive 70/157/EEC, available at `http://ec.europa.eu/enterprise/sectors/automotive/documents/directives/directive-70-157-eec_en.htm`.

[44] W. Rachele, Object Pascal with Delphi, Wordware Publishing, Inc, 2000.

[45] S. J. Hanson, R. R. Rosinski, Programmer perceptions of productivity and programming tools, Communications of the ACM 28 (2) (1985) 180–189.

[46] J. Luoma, S. Kelly, J.-P. Tolvanen, Defining domain-specific modeling languages: Collected experiences, in: Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM), 2004.