applied sciences

MDPI

Article

A Tool Support for Model-Driven Development: An Industrial Case Study from a Measurement Domain

Tomaž Kos [1], Marjan Mernik [2] and Tomaž Kosar [2,*]

[1] DEWESoft d.o.o., 1420 Trbovlje, Slovenia; tomaz.kos@dewesoft.com
[2] Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia; marjan.mernik@um.si
* Correspondence: tomaz.kosar@um.si; Tel.: +386-220-7448

Received: 5 September 2019; Accepted: 22 October 2019; Published: 26 October 2019

Abstract: End-user programming may utilize Domain-Specific Modeling Languages (DSMLs) to develop applications in the form of models, using only abstractions found in a specific problem domain. Indeed, the productivity benefits reported from Model-Driven Development (MDD) are hard to ignore, and a number of MDD solutions are flourishing. However, not all stories from industry on MDD are successful. End-users, without having software development skills, are more likely to introduce software errors than professional programmers. In this study, we propose and encourage other DSML developers to extend the development of DSML with tool support. We believe the programming tools (e.g., debugger, testing tool, refactoring tool) are also needed for end-users to ensure the proper functioning of the products they develop. It is imperative that domain experts are provided with tools that work on the abstraction level that is familiar to them. In this paper, an industrial experience is presented for building various tools for usage in MDD. Debugger, automated testing infrastructure, refactoring, and other tools were implemented for Sequencer, a DSML. Our experience with the implementation of tool support for MDD confirms that these tools are indispensable for end-user programming in practice, and that implementing those tools might not be as costly as expected.

Keywords: Model-Driven Development; Domain-Specific Modeling Languages; tool support; debugging; model-based testing; usage experience; measurement systems; data acquisition

1. Introduction

We live in a time when demand for the development of software and extending of the existing ones are increasing enormously. Computer Science is, therefore, seeking for solutions on how to increase programmers' productivity in many different perspectives. One of the promising solutions are high-level abstraction languages, often referred to as Domain-Specific Languages (DSLs) [1]. DSL captures the essential characteristics of a problem space in a manner that is decoupled from the details of a specific solution space [2,3]. In other words, programming in DSLs concentrates on what the software should do instead of how the software should do it.

A twin brother of DSL is Domain-Specific Modeling Language (DSML) [4] where, instead of using textual concepts, we use visual concepts. Hence, the textual program is substituted by a model which becomes a first-class concept in the development process. Domain experts, the end-users of DSMLs, are primarily developing models, and such a development process driven by the model is known as Model-Driven Development (MDD). The broader term is Model-Driven Engineering (MDE), a software development methodology that goes beyond the development of models supporting other

activities and tasks to a complete software engineering process. The purpose of models is different according to the problem domain. It is essential that abstractions in models are coined from the domain knowledge and perfectly understandable to domain experts. The benefits reported from MDD are flexibility, productivity, reliability, and usability [5–7], which have been shown through empirical evaluations [8,9], and a number of MDD solutions are flourishing [10–14]. Another benefit of MDD is that it enables end-users and domain experts to program without having professional software knowledge, but they can express themselves in DSMLs. Scaffidi et al. showed that the number of end-users in the last years has been increasing [15].

MDD in industrial environments does not only feature in success stories, but also in failure stories [16]. Indeed, good results are not always achieved with end-user programming, despite the predictions [17]. Domain experts, without having software development skills, are more likely to introduce algorithmic errors than professional programmers, because they lack software development training and proper tool support [18]. We believe that the proper programming tools (e.g., debugger, testing tool, refactoring tool) are needed for domain experts to ensure the integrity of the products they develop [19]. With a large pool of domain experts and the rising cost of software failures, it is imperative that DSML domain experts are provided with tools that allow them to detect and find software errors at an abstraction level that is familiar to them. The utility of a new DSML is seriously diminished if supporting tools needed by the domain experts are not available.

The design and implementation of a DSML can be challenging and expensive [2], even if we use tools for MDE (EMF [20], MetaEdit+ [21], Acceleo (http://acceleo.org/), GEMOC [22], etc). It requires knowledge of modeling a language, implementation of a code generation, as well as domain knowledge. Currently, there are only a few tools for MDE that support generation of tools like a debugger [23,24]. Implementing a DSML tool from scratch can be more time consuming, error-prone, and costly. This is why, in most cases, a Model-Driven Development Environment (MDDE) is provided without tools like a model debugger, but, as can be observed in the case of many programming languages, like Haskell [25], the lack of debuggers and profilers, inadequate support by Integrated Development Environments, and poor interoperability with mainstream languages, can be contributing factors for the resistances of DSML within the software industry [26].

As stated before, MDDE does not have tool support by default. We believe that this prevents domain experts from being productive and decreases the general public's interest in MDE and MDD. In our case, we firstly implemented a DSML called Sequencer [27] without tool support, but, from the experience working with customers, it was evident that Sequencer itself was not sufficient, and that domain experts need additional help during their development. Implementing several tools (debugger [28], automated tests [29], refactoring, generation of (template) models, etc.) helped domain experts to be more efficient and satisfied with the development of measurement procedures.

Until now, there has been little evidence on providing DSML tool support [30–32]. In this paper, we address that gap by reporting on a successful industrial study on implementing a Ladybird (a DSML debugger), SeTT (automated testing for DSML programs), and RTool (a DSML refactoring tool) for the measurement domain. Our experience with the development of these tools confirms that MDD tool support is indispensable for acceptance of MDD by domain experts. Besides this lesson learned in the industrial environment, we report our experience with end-users, maintaining and evolving the DSML, which other practitioners may also apply after introducing DSML in a real-world environment. We also report on the implementation and maintenance effort for the development of DSML with the mentioned tools.

The remainder of this paper is organized as follows. Section 2 discusses the motivation for the work. Section 3 presents the measurement domain with a description of DSML Sequencer. The generalized ideas for implementation of the debugger, testing, and refactoring tool are described in Section 4. Tool support for MDD is illustrated in a real case scenario in Section 5. A discussion with our experience implementing tool support for MDD follows in Section 6. The related work on the

debuggers, testing, and other MDD support aspects are discussed in Section 7. Finally, the concluding remarks are summarized in Section 8.

2. The Motivation for an Experience Report

In the systematic mapping study in the field of DSLs [33] which included 476 studies from 2006 to 2012 from 4 different digital libraries (ISI Web of Science, ACM Digital Library, IEEE Xplore, and Science Direct), it has been shown that researchers are mostly interested in the development of novelty techniques (82%). Only 5.7% of the studies were connected with the evaluation of their approach, whilst 10.1% of the studies were experience reports. We believe the acceptance of DSMLs by the industry will not increase with the development of the DSML novelty techniques. Strong support from empirical evidence can do that.

In a study by Czech at al. [34], the authors searched for best practices from MDD. Although searching in four digital libraries (ACM Digital Library, IEEE Xplore, Science Direct, and Springer Link), meta-search engines (Google Scholar), snowballing the papers, the authors found only 21 studies that shared experiences, patterns, lessons learned, or good practices. We believe this is not enough for practitioners from different fields and problem domains. We can also not expect that the industry mainstream will adopt our ideas and use MDD without clear and good examples reported in the research literature.

The same study [34] also reported 321 best practices in 21 studies. The large majority of practices (75%) were reported only once, clearly, showing that there is no consensus on what are the best practices in the research field of DSMLs. DSML researchers will have to concentrate on this aspect of research in the future.

There are three different motivations for this paper that are connected with the above observations. Firstly, as stated in the title of this Section, this is an experience report. It is a result of our long-lasting interest in the field of DSLs and DSMLs. Working with Sequencer's customers for over 10 years, in this study, we report our experience from the real-world environment. Secondly, we want to add our advice to the list of best practices. We do not want to bring more confusion with a dozen new best practices. Our simple message is that DSMLs need tool support for being useful in the industrial environment. The third motivation for this paper is to convince others that the shortcomings (investment in the development of DSML, etc.) do pay off when the end-user can develop their applications in new defined DSML very quickly.

3. The Measurement Domain and Sequencer

A domain-specific language is written for a specific purpose or domain. In this Section, we will try to illustrate what is the domain of the language, who are the users, and in which industry Sequencer is used.

Sequencer was developed in the DEWESoft company, which is developing software for Data Acquisition, Data Processing, Data Analyzing, and Data Storage (http://www.dewesoft.com/). DEWESoft generates 30% of the income in the Automotive market. Other key customers come from Aerospace and the Railway industry, with 20% income each. DEWESoft products are even used in the Space industry. NASA awarded the DEWESoft product Krypton as "Product of the month" in 2019 [35].

DEWESoft software represents an intermediate stage between general measurement devices (which are usually very difficult for the user to use, e.g., oscilloscope, spectrum analyzer), and measurement systems specially adapted for specific devices. The DEWESoft software package can capture different data, like analog and digital signals, CAN data, GPS data, and many more. Despite features that allow easier usage, the measurements are very demanding, and require precise definitions of the data acquisition procedure. In order to enable end-user programming, a DSML called Sequencer [27] and its MDDE have been developed (Figure 1).

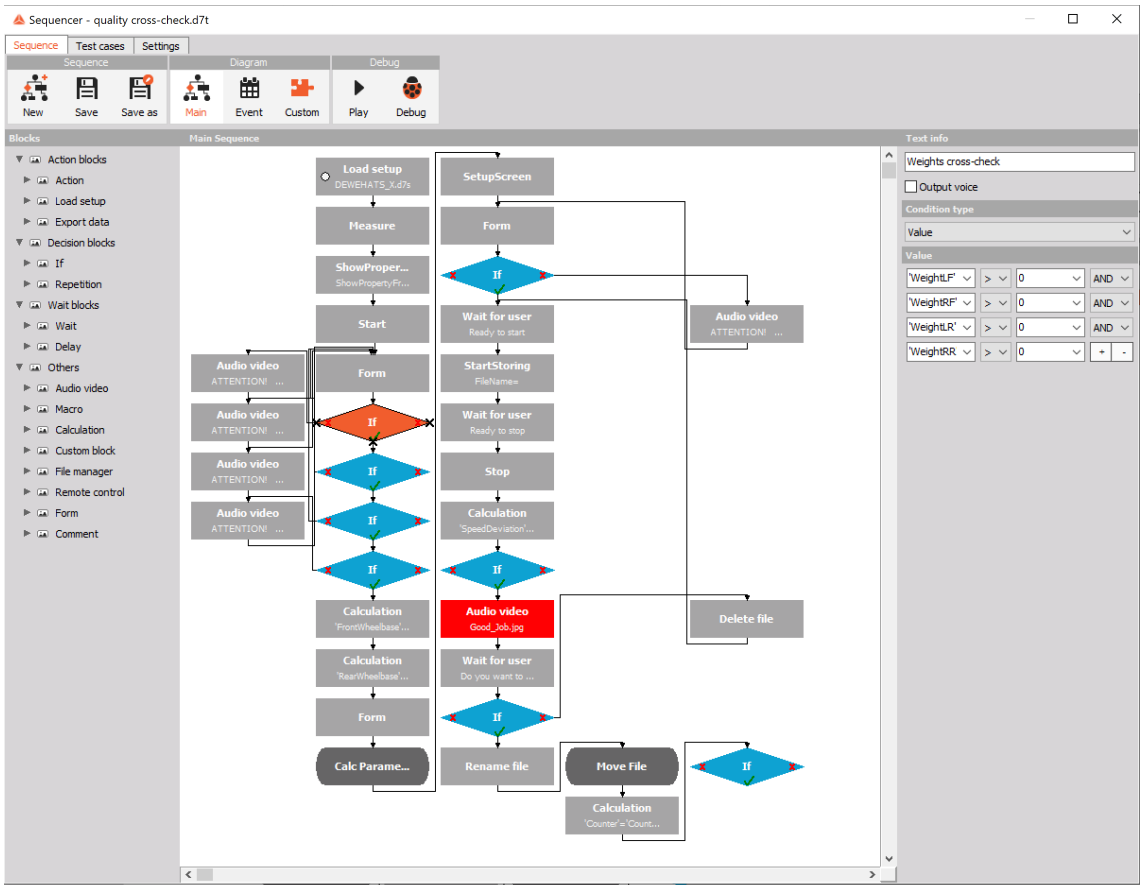

Figure 1. The DEWESoft's modeling environment.

In Sequencer, domain experts create a measurement procedure using predefined modeling blocks (see the left side of MDDE in Figure 1) which represent actions in the measurement system. Modeling blocks also contain local and global variables (that represent channels on measurement devices, sensors, etc.). Their purpose is to store values within the measurements (see the right side of MDDE in Figure 1). An example of a measurement procedure is shown in Figure 1 (in the centre of MDDE). This is a measurement procedure for tires' quality cross-checks from the automotive industry. The procedure controls the data acquisition during the measurement, and shows current values on a professional driver display (see Figure 2). The professional driver observes (see Figure 2) the current measured values, like different velocities, direction, acceleration, GPS coordinates, etc. After manoeuvring, the metrics for quality cross-check are calculated, displayed on the screen, and stored.

Note, that the MDDE in Figure 1 shows the "Main model". Besides, we can define "Event" and "Custom" models (see the top of Figure 1). The latter is similar to "functions" in programming languages, and was incorporated to have better control over other models' extent and re-usability purposes.

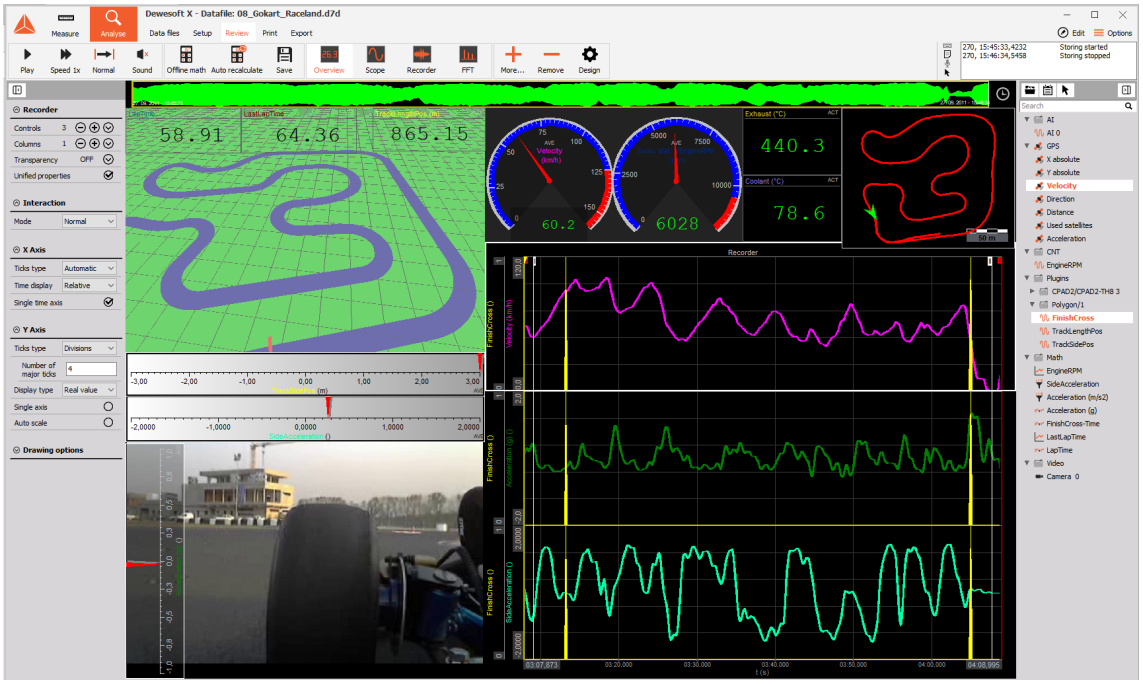

Figure 2. Execution of measurement procedure developed in Sequencer.

4. Proposed Method: Tool Support for End-User Programming

Although with Sequencer the programming measurement procedures have been simplified and shifted to the higher abstraction layer enabling end-user programming, the domain experts reported several shortcomings, like:

- It is difficult to find bugs during very complex measuring procedures, because bugs can occur in the model or the measuring equipment. *Step-by-step* execution would be helpful, since it would improve control over the execution.
- It is often needed to *stop execution* before or after a specific building block during the measuring procedure. In some cases, stopping the measuring procedure at a specific trigger is desirable. Especially for those applications that are time-critical (e.g., the temperature rises to a certain level when testing the water heater).
- The ability to monitor and alter the *values* of a running measurement procedure would be beneficial for testing.
- An *animation* of the measuring procedure would improve the domain expert's understanding.
- It is harder to monitor the measurement procedures simply by displaying channels on the visual instruments/graphs or looking at the models.
- The measurement procedures are usually executed very fast. Consequently, tracking measurement results (*printing*) and understanding heavy calculations during the measurement is difficult.
- Some advanced domain experts wanted to test measurement procedures by the white-box method (*stack traces*).
- *Testing* the same measurement procedures and comparison with expected results is often repeated.

From the above description, it is obvious that domain experts are asking for additional support during the development of measurement procedures. The following subsection presents the tools which were combined to enable tool support for end-user programming.

4.1. Debugging Models in Sequencer

To enable debugging features, the architecture of the Sequencer has been extended as denoted by the darker shapes in Figure 3. The front-end contains three levels. At the modeling level, we

can find Sequencer with different models. A Sequencer is connected with the execution model in the back-end [27]. The execution model is an internal representation of all Sequencer models, and is responsible for communication with the measurement framework (domain framework) that further communicates with the acquisition equipment. To support debugging front-end features, the execution model has been extended using different components [28]: Source Model Mapping, Debugging Actions, Execution Responses, and Memory Mapping.

The above components are building the internal structures and representations needed by different Ladybird (a debugger for Sequencer) features:

- Execution modes,
- Steps,
- Breakpoints,
- Print statements,
- Model animations,
- Variable views, and
- Stack traces.

The main purpose of Source Model Mapping is to connect the execution model code with the modeling block from a DSML model. Debugging Actions is a communication interface between debugging features and the execution model. Besides, Debugging Actions controls the execution of model slices (small units of the execution model). Execution Responses is a message system between the debugging environment and the execution model. The measurement memory is accessible through Memory Mapping, where the user can trace the measurement values during debugging. At the last application level, the front-end runs the measurement procedure in normal or debugging mode.

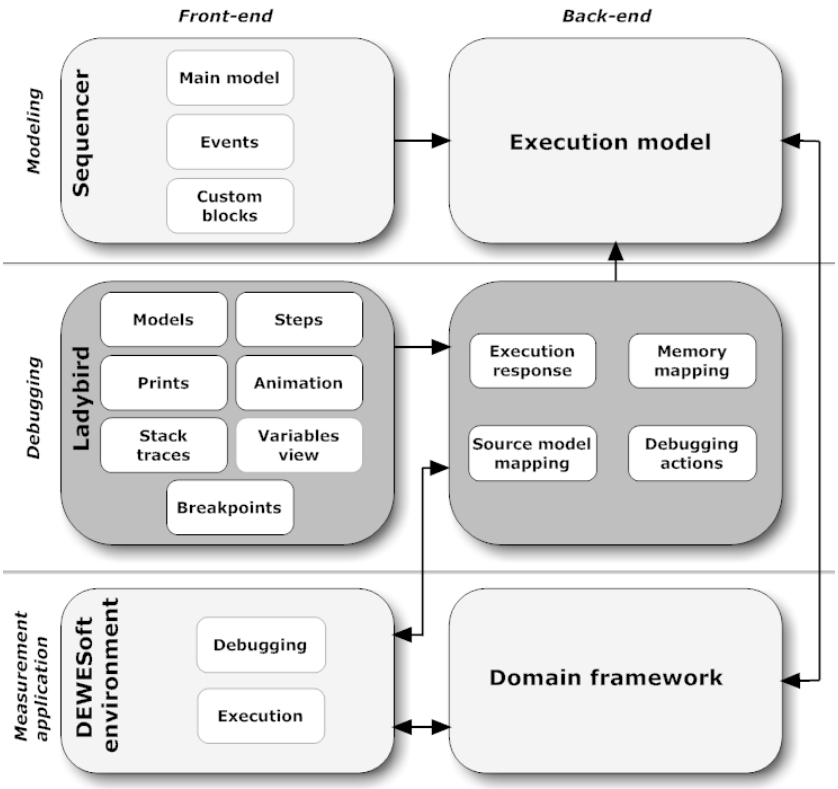

Figure 3. Debugger support in DEWESoft.

A detailed implementation description of LadyBird can be found in [28].

4.2. Model-Based Testing of Measurement Procedures

Testing of the acquired data is extremely demanding in any measurement system. There are several reasons for that. One of them is that data acquisition is a complex combination of software, electrical and mechanical components. These parts need to be synchronized throughout the whole measurement process. A lot of testing with repetitions is needed to be confident in a whole measurement package. A logical next step for DEWESoft was to automate the testing of measurement procedures [29].

In textual programming, we use unit testing to automate the process of testing and gain the possibility of running the tests over and over again. This technique is based on the idea of testing parts of a program and comparing the obtained results with those expected. The same idea was applied to the SeTT (Sequencer Test Tool) [29]. The main difference between traditional unit testing and SeTT are testing artefacts. Since measurement procedures in the Sequencer are models, SeTT tests models. For tests we could use textual notation similar to [36], however, DEWESoft customers are used to developing their measurement procedures with models, therefore, we decided to support the development of measurement tests using Model-Based Testing (MBT) [37].

From the DEWESoft user's point of view, we wanted to separate the development of tests completely from measurement procedures. Therefore, DEWESoft users do not have the testing support by default in their MDDE. They have to switch to the testing environment to define tests. However, we kept the same notation for models in MBT as used in the measurement procedures. All building block and connections from the MDDE can also appear in the MBT environment. The difference between the MDDE and MBT environment is that, later, we included support for testing in the form of a new build block, "Assert", that tests the values from measurements. There are several options to define the structure of a DSML (grammar, metamodel, etc). The Sequencer notation was defined by a metamodel [38–40]. SeTT's notation is derived from the Sequencer's metamodel, with the extension—the Assert building block (Figure 4). Our implementation of SeTT is a non-invasive extension of a DSL [41,42]. In this case, Sequencer kept the original notation. Note that non-invasive DSL extension is possible for those DSLs where new concepts are orthogonal to base DSL concepts. In our case, the Assert concept did not require any changes in other concepts from the base metamodel.

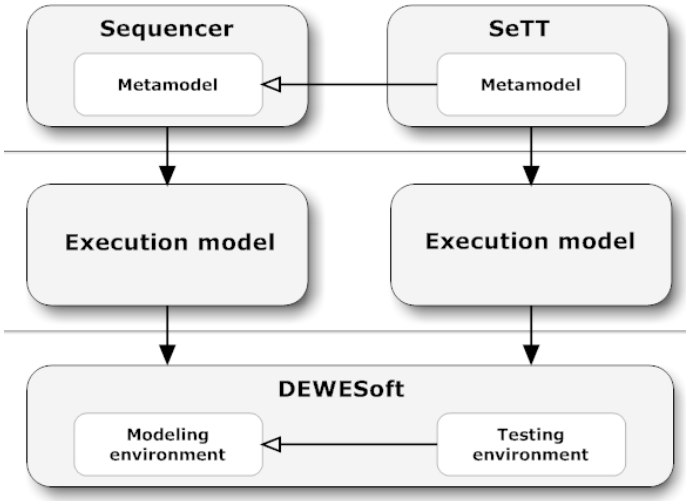

Figure 4. Relation between Sequencer and SeTT in DEWESoft.

4.3. Refactoring Measurement Models

In refactoring, we perform actions which restructure the programming code with a desire to make the code more efficient and effective [43]. By improving the design, the code's behavior should remain unchanged. Transformation can be done manually, however, modern Integrated Development

Environments enable tool-driven solutions. Since refactoring tools help automate tedious design changes, they are highly desired by programmers.

In modeling, the meaning of refactoring is similar to programming code, except for the transformation target—instead of the code, the model is the object of transformation [44]. Within the Sequencer, domain experts are enabled with the possibility of improving models automatically. We agreed to support those refactoring methods where we can guarantee to preserve the model's behavior. In some refactoring methods, predictions of the model changes are clear [45]. We must understand that our domain experts are not programmers, but domain experts. Incomplete code after transformations where further manual actions are needed could cause problems to our domain experts when preserving the models' behavior. Currently, the following refactoring methods are supported in Sequencer: Extract custom block, inline extract custom block, rename custom block, add a parameter, delete parameter, rename parameter, rename local variables and rename device name. The meanings of those methods are similar to those defined by the programming code [46].

From the point of view of implementation, our refactoring algorithm works in several steps (see Figure 5). First of all, the user has to select artefact (modeling block, custom block, group of modeling blocks, etc). "Input artefact" which is still in visual notation is given to RTool, where we transform the visual notation into an internal artefact representation. Further, RTool transforms the artefact based on the chosen refactoring. Each type of refactoring uses its transformation and each transformation is implemented separately. The result of the transformation is an "output artefact". The next step is to weave the artefact back to the model. Then, we check if the new model corresponds to the visual notation of Sequencer (if the new model corresponds to the metamodel). The last step (to check the behavior of the new model after refactoring) is optional but recommended. Using MBT ensures unchanged behavior of the measurement procedure. The testing tool SeTT (described in Section 4.2) is used to test the new model's behavior.

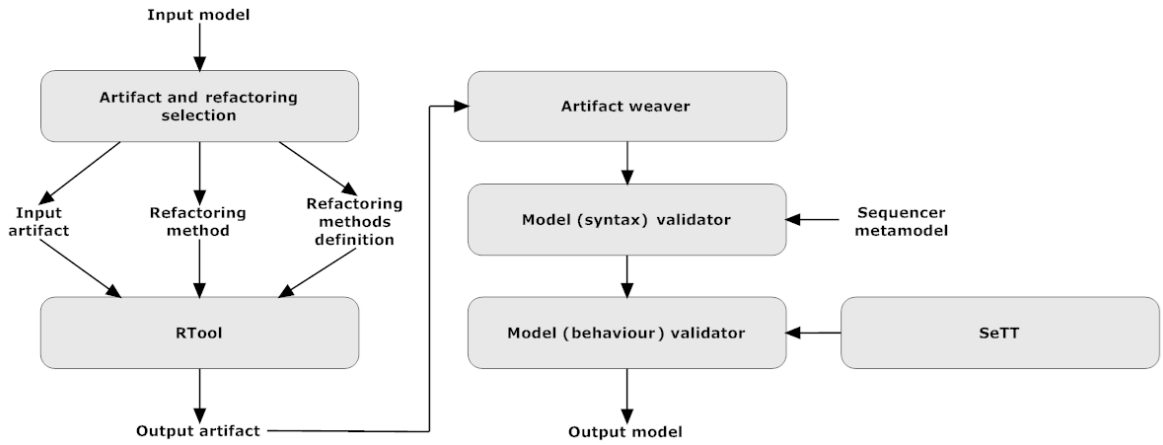

Figure 5. Model transformation for refactoring in Sequencer.

5. A Demonstration of Tool Support: Refactoring a Model

As an example, we will take the measurement procedure that checks balloon elasticity during several consecutive inflations and deflations (https://youtu.be/ofCSJZu9NxE), and demonstrate refactoring inside Sequencer. We intentionally took refactoring as an example, since someone could argue that such small domain-specific models do not need refactoring support, nor MBT as a supplemental feature.

In a balloon elasticity test, in each repetition, we inflate the balloon until at a certain pressure (Pmax), hold the pressure for a certain time, deflate the balloon until a certain pressure (Pmin), and hold the pressure again. This procedure is repeated 1000 times. The measurement procedure for the balloon durability test described above was written by one of the DEWESoft's customers domain experts. First, he/she decided to divide operations into three custom blocks: Inflation (inflating the

balloon), locked (idle mode of the balloon), and deflation (deflating the balloon). Then, the main procedure presented in Figure 6a was written, where the first calculation block sets the counter of the inflation/deflation procedure to zero (field "int_cycles"). The first "if" block checks if we have reached 1000 cycles of inflation and deflation. The next "if" block checks whether the pressure is below that desired (Pmax). If this is true, we call custom block "inflate" and, after inflation, check again if we have reached Pmax. When the pressure is reached, we call custom block "locked" to hold the pressure, and the following action, block Delay, does that for 3 s. After that, we start with deflation (custom block "deflate") and check if the pressure reaches Pmin with the next "if" block. After deflation, we hold the pressure (we call the custom block "locked") and delay for 3 s. The last block increases the counter of the inflation/deflation procedure and returns it back to the first "if" modeling block to check if we reached the desired number of test repetitions.

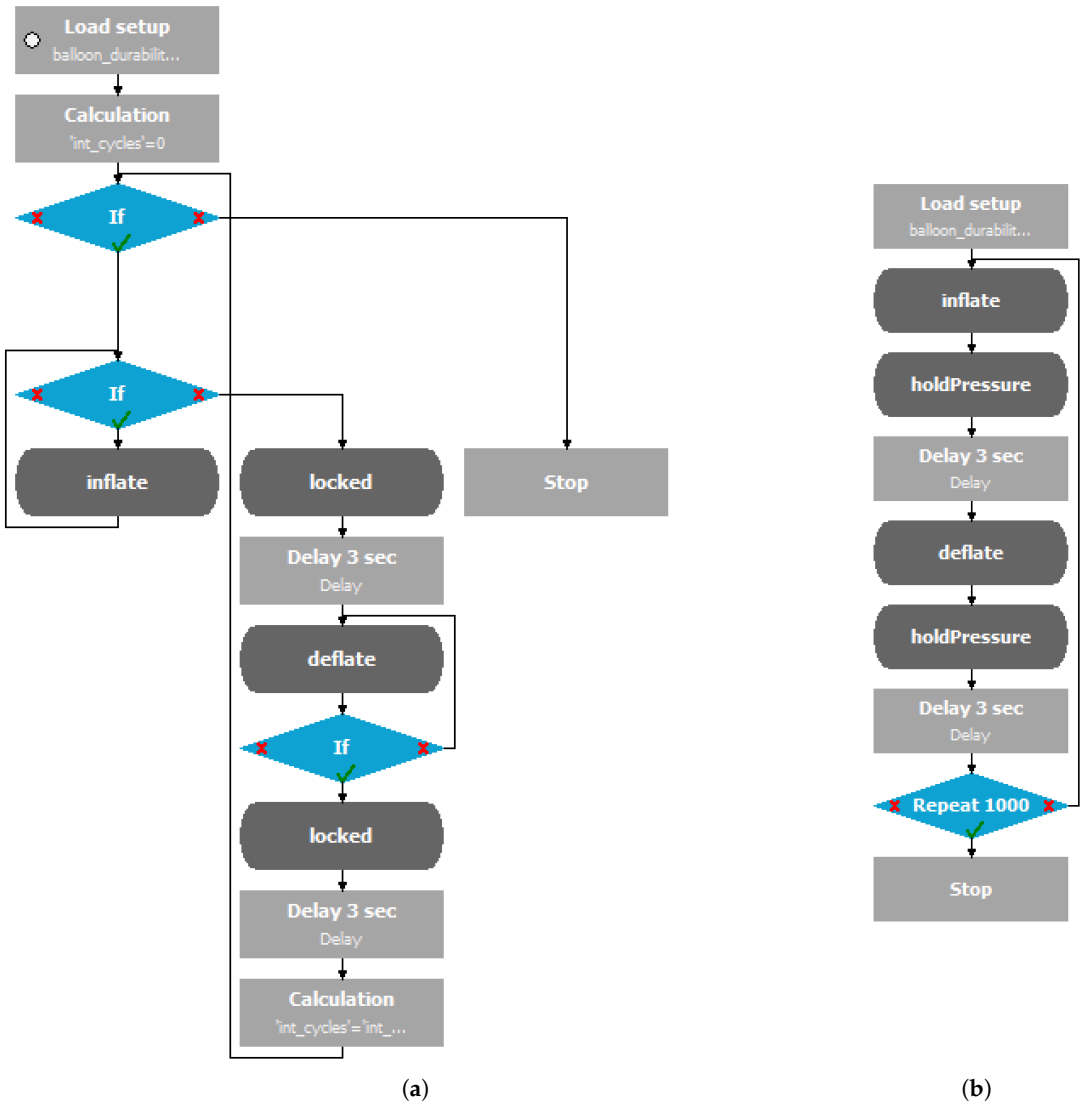

Figure 6. Balloon durability test—(**a**) Before refactoring (**b**) After refactoring.

At first glance, this measurement procedure in Sequencer was completely unclear to us. The problem with this procedure is that it has a lot of "if" blocks, which complicate the understanding of the flow in this sequence. We agreed that those "if" blocks that check the pressure (Pmax and Pmin) actually belong to the custom blocks "inflate" and "deflate". To perform these changes we first used "Inline custom block", which moved all the blocks from custom block "inflate" into the main model and deleted this custom block. We did the same for custom block "deflate". Then, we selected those blocks

previously moved back to the main procedure, together with the corresponding "if" block, and used the refactoring method "extract custom block" and created new custom blocks "inflate" and "deflate". Sequencer also supports the very basic refactoring method "rename custom block". This method was used for renaming the custom block "locked" to "holdPressure".

Next, we wanted to replace the "if" block that checks the counter (int_cycles) of repetition with the "repetition" block. To replace the repetition functionality we had to remove both calculation blocks from Figure 6a (that maintain the counter of cycles), remove the "if" block at the top of the measurement procedure, and place repeat "block" at the bottom of the measurement procedure. For all these operations there is no such refactoring support in Sequencer. Therefore, this refactoring has to be done manually. For the purpose of manual refactoring it is necessary to use an MBT tool SeTT. After manual refactoring execution, we performed the manual refactoring explained, above and copy-pasted the sequence into the MBT environment. We added several Assert building blocks at the end of the measurement procedure to check the behavior on a new sequence. Fortunately, SeTT returned no errors/failures, so our manual refactoring was done correctly.

Figure 6b presents the final version of the measurement procedure for the balloon elasticity test after automatic and manual refactoring. Observe that the flow of this measurement procedure is quite simple and easy to understand if we compare it to the initial one from Figure 6a.

6. Our Experiences

Ten years of industrial experience in using Sequencer confirm that domain experts have been able to program their own measurement procedures and change them during measurements with little support from DEWESoft software engineers. In the next subsections, we are highlighting several factors that enabled better acceptance of Sequencer. We also report experience with the development of Sequencer and implementation effort.

6.1. Growing the Community

The end-user community using Sequencer has been growing each year. The need for building a strong community system (https://training.dewesoft.com/) was one of the first logical steps. Support was added for online, in-house and on-site training, as well as webinars and other features.

Up to now, more than 400 new domain experts have attended workshops, and an additional 150 have completed the online training. Two-thirds of them had never programmed in general-purpose languages, and they are using Sequencer in their daily work. Over recent years, domain experts have reported increased productivity in terms of time spent developing a measurement procedure. Moreover, the number of measurement procedures has increased. Just within the company DEWESoft, over 5000 measurement procedures have been created, which are used daily with automated tests in continuous integration. For such a scope of work, we would need several years of work. Without domain experts being able to program in Sequencer, small DEWESoft development team would not be able to assist such a fast-growing community.

6.2. Close Collaboration with End-Users

New Sequencer features (breakpoints, animations, etc.) enabled domain experts to eliminate errors without interference from the DEWESoft software engineers. From the example of the balloon durability test, we could see the obvious need in Sequencer for a refactoring tool. Even within a relatively small domain, the domain experts have the freedom to model their measurement procedures in various ways. The refactoring tool is indispensable if we want to improve the structure of the DSML models.

One of the most experienced end-users who has written 900 sequences so far, outlined in an interview that, out of the three tools mentioned in Section 4, he is using SeTT and RTool regularly. The debugger is used only with severe errors with more complex sequences. He adds that many times the problem is not in the tool, but in the understanding of the Sequencer, and sometimes he has an

incorrect design of a measurement procedure. In those cases, a deeper discussion was needed with the DEWESoft software engineers to understand the underlying framework and solve the issue.

6.3. Raising the Level of Domain Abstraction

The Sequencer domain is pretty small, having only around 20 different modeling blocks, on average eight properties per block, three types of models (main, events, and custom blocks), etc. One would think the language is pretty simple, since the largest measurement procedure consists of 451 building blocks.

However, the underlying framework used in Sequencer is pretty huge, containing more than 1700 kLOC, many possibilities are given, and the modeling language can be used in different domains, besides automotive, also in aerospace, acoustic, power & energy, and noise & vibration, as shown in Table 1. Even one domain can be very complicated. For instance, some typical examples of measuring procedures in the automotive domain are pass-by noise test, double lane-change test, functional safety test, lane departure test, car brake test, harmonic monitoring, component testing, wind tunnel testing, modal analysis, balancing etc. All these tests can have numerous different variations. For all these domains we currently have 1712 model-based in-house tests.

Therefore, it was very useful to fine-grade automated tests to specific domains. Whenever we have on-site testing at a customer's workplace and want to check the measurement equipment, we run automated tests from a specific domain. The current number of specific domain test cases is shown in Table 1.

Table 1. Domains and number of in-house automated tests.

	Test Cases
Automotive	61
Aerospace	39
Acoustics	51
Power & energy	385
Noise & vibration	1176
Sum	1712

6.4. Relying on Model-Based Tests in Continuous Integration

DEWESoft's customers are relying on accurate data from measurement equipment, and sometimes there is no chance for repetition of tests. Incorrect data, or even latency in measurements, can cost millions of dollars. Therefore, it is very important to eliminate as many system errors as possible during the development phase.

As shown in Table 2, over 5000 measurement procedures have been created, which are used daily in continuous integration. Besides 1712 domain tests, as shown in Table 1, we are also testing hardware (analog cards, amplifiers, GPS receivers, etc.), drivers, front-end, and back-end on 24 different measurement devices. Same test cases are executed multiple times (column Executions in Table 2) for different systems (Win32, Win64, etc.), DEWESoft versions (beta, release candidate, release), etc. The last execution we checked on 3 October 2019 contained 18,371 executions of model-based tests, which, on average, contain 2.6 assert blocks per test case. The average time for the whole suite for all domains is around 4453.5 min (approximately 3 h per each measurement device).

Table 2. Running test cases in DEWESoft continuous integration system.

	Test Cases	Executions	Execution Time (min)
Domains	1712	6895	1528.5
Hardware and drivers	2496	8166	1615.5
Front-end	99	396	360
Back-end	777	2914	949.5
Sum	5084	18,371	4453.5

6.5. Development of MDDE from Scratch

We developed the first metamodel in GME [4]. After providing a first modeling notation, we checked the generation of the code by a tool, and there was no support for Delphi in which DEWESoft is written. We searched for other workbenches that support generation of code in that language (we eliminated source-to-source transformation and similar solutions, which would complicate further development of our tool). Secondly, the integration of DSML with DEWESoft (a massive product containing over 900 kLOC back in 2009) was challenging. One of the requirements was that DEWESoft remains in one form. Running an external modeling application was not an option, due to the possible effect on the speed of measurement (note that data acquisition is running in a loop, which must collect data without any latency, specific frequency, without data loss).

Tools for MDE (e.g., MetaEdit+, GME, EMF) are useful and powerful tools when you start with a new product. Having a software, it takes a lot of adjustments and sacrifice if one would like to connect modeling with existent MDE workbenches. In our case, facing many obstacles, we decided to start developing MDDE from scratch.

6.6. Removing Features from MDDE

We strongly believe that not every tool is equally important for different DSMLs. Domains and end-users are different, and can results that specific tool is important for one DSML but not useful for the end-users of the second DSML. As stated in the related work section, there are studies where support for multi-view development was shown to be useful for different domains (e.g., mobile application development [47]). As stated in the related work section, we experienced having model-based as well as code-based notation in Sequencer at the same time [48]. Although, the text-based notation had our full support in terms of documentation, DEWESoft's help desk, etc. it was not the first choice of our end-users. Many of them never even switched to the textual environment, which at the end, due to expensive maintenance, resulted in the withdrawal of text-based notation from Sequencer.

From our experience, we can conclude that not every tool for MDDE will be accepted by end-users. However, developers should not stop on seeking new solutions to support end-user development.

6.7. DSML Development Effort

The development effort for Sequencer with tool support is presented in Table 3. Due to customer support and maintenance, these tools are still being developed today. Currently, the entire MDDE comprises about 35k LOC (see Table 3), and has been developed and maintained over the last 10 years.

Table 3. Development effort for Sequencer with tool support.

	Debugger	SeTT	RTool	Sequencer	Sum
kLOC	3	4	1	27	35
person months	3	6	1	12	22

Looking from this perspective, we can say that the development of such tools has taken a lot of effort. However, DSMLs are usually part of a larger system. It is the same with the Sequencer.

A DEWESoft product comprises 1.712 kLOC, which means that the Sequencer represents only 2% of the total product. From this perspective, the development of DSMLs with associated tools represents a relatively small investment with high added value to the whole product.

6.8. Maintenance Effort for Sequencer with Tool Support

One of the arguments often heard about DSML is that there is more effort needed to maintain and evolve DSMLs. We looked in our ticketing system, and provide development statistics for the entire product, as well as for Sequencer, in Table 4. The number of all ticket, questions, bugs, wishes, resolved issues is approximately the same as the ratio that Sequencer represents in the whole system (column kLOC in Table 4). This experience shows clearly there is no difference between maintenance effort for DSML and the rest of the system.

Table 4. Comparing maintenance effort between DSML and rest of the system.

	Tickets	Questions	Bugs	Wishes	Resolved Issues	kLOC
DEWESoft with tool support	15,729	6299	4173	5257	7718	1712
Sequencer	226	106	48	72	100	35
Ratio	1.4%	1.7%	1.2%	1.4%	1.3%	2.0%

7. Related Work

End-user programming [49] refers to individuals who are not professional software developers, but are conducting some kind of programming during every day professional activities. Even for end-users, numerous studies were done, where it was argued that the end-user programming should be provided with tool support. Spreadsheet languages are a nice example of end-user programming, and Jannach and Schmitz [50] have shown how important debugging support is [50].

Another group of programming language researchers is concentrated around block-based programming, where programming commands are represented mainly as visual blocks [51]. Block-based programming languages have been applied successfully as an introduction to novice programmers. These block-based programming languages are fun and motivating to start programming computer games (Scratch [52]), interactive multimedia (Pixly), robotic control (RoboScratch), mobile phone application development (MIT AppInventor), etc. Recently, researchers have also highlighted the need for debugging in block-based programming [53].

End-users, novice programmers, and domain experts share limited knowledge of programming. However, these groups of programmers should have appropriate comprehension tools in their programming environments.

In the remainder of this Section, we present a recent effort done in the field of Tool Support for MDD, and compare it with features in Sequencer.

7.1. Summary of Related Work in Debugging of Models

First, we have to distinguish between *Model-Based Debugging* and *debugging a model*. In Model-Based Debugging, a *model* is built that reflects the behavior of the (potentially incorrect) program. On the other hand, debugging a model observes and controls the execution of a model and provides a behavioral representation. A programmer, while debugging the programming code, constructs a mental representation of the code. Many different aspects [54] have to be taken into account when debugging programming code. Many models for Model-Based Debugging of textual programming code have been proposed to facilitate programmers cognitive load [55]. Debugging techniques have been researched and developed extensively for programming languages. Model-Based Debugging is a well-established research field, but when it comes to debugging a model, very few debugging tools and techniques are available [32]. More generally, debugging a model in the context of MDD is not as mature as Model-Based Debugging of programming code.

Some of the earlier studies tried to understand if a relationship exists between debugging of programming code with models in MDD [30]. In this context, Mannadiar and Vangheluwe proposed a mapping of the most common debugging concepts, from programming languages to DSMLs. Programming "debugging" 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) were mapped to concepts in a DSML debugger. Mapping distinguishes between two different facets: The debugging of model transformations and the debugging of domain-specific models. Mannadiar and Vangheluwe presented a prototype of debugging concepts with a tool, ATOM3 [30]. Debugging in LadyBird supports an almost identical set of primitives as in Mannadiar and Vangheluwe's work.

Sun and Gray [31] presented an MTBD Debugger, which is a model transformation debugger based on the MTBD execution engine. MTBD Debugger enables users to step through the transformation execution process and track the model's state during a transformation. Similarly, SeTT enables a very basic transformation during execution. Both tools offer debugging on a higher level of abstraction, hiding low-level execution details during the debugging process.

Van Mierlo in his PhD [32] exposed current research problems in the field of Debugging a model in MDD, where he suggests that dedicated debugging support is required for DSMLs and workbenches. His work concentrated on providing a structured approach for turning modeling and simulation environments into interactive debugging environments [32]. He applied his approach to build debugging environments for different general-purpose modeling languages (statecharts, Petri nets, etc.) as well as DSMLs. By building such a set of debugging environments, he demonstrated that his approach is generally applicable for any DSML.

7.2. Summary of Related Work in Model-Based Testing

MDD has not only changed the way software is built, it also brings the new way this software must be tested [56] – MBT is recommended for model-driven development. Mussa et al. [56] have surveyed model-driven testing techniques. They concentrated in the following aspects of a model-driven testing: (1) Modeling language: What kind of modeling language researchers and practitioners are using for MBT, (2) Automatic test generation: If the MBT systems provide functionality of automated generation of tests, (3) Testing target: The artefacts of testing (models, implementation, etc.), (4) Tool support: MBT tools that can be used to reduce time to market to generate the test cases. Another survey on MBT was done by Uting et al. [37].

Most commonly, researchers use UML notation as a modeling language for MBT [57–59]. It is similar with the tool Tedeso [60], which is a testing platform integrated within Eclipse and used by Siemens. It uses UML-based notation to specify a test which, during the generation phase, is connected with a specific unit testing API (JUnit, NUnit), depending on the selected target language. Similarly, Sequencer has its domain-specific notation, which uses DUnit API for code generation. A Tedeso supports automatic test generation, while SeTT does not have such a feature. The Tedeso testing target is the more extensible and interoperable solution, available also for general-purpose languages, whilst SeTT is dedicated to Sequencer, and hardly-coupled within the MDDE.

An example of a tool with automatic test generation in the form of a test model, U2TP (UML 2.0 Testing Profile) was introduced by Elallaoui et al. [61]. They automated the transformation of design models into test models and generating test cases. As stated earlier, Sequencer does not support the automated generation of SeTT models from Sequencer models. However, automated generation is possible, since SeTT's metamodel is an extension of Sequencer's metamodel. Consequently, all Sequencer models are valid SeTT models. The missing part which should be added is the testing code in models. Our interest in an automated generation is in mutation MBT [62]. Here, part of the model is mutated (changed), and new test cases are executed to check if we can find any error in the measurements. Note, that the measurement domain is a complex system that integrates mechanical,

electrical, and software subsystems. It takes a lot of repetitions with different settings to be sure that the solution is working properly. With mutation MBT we could generate missing manual model tests.

Model-Based Testing solutions target different software artefacts, ranging from abstract [57] to executable models [58], from specific [63] to composite parts of implementation [59], or even target the whole system [64]. In the case of SeTT, we can test executable models as well as the whole system. At the customer location, we are opting for testing the whole measurement system. As such, we can validate if the measurement system (equipment together with measurement models) is installed properly, and that the real-world data acquisitions can be started. For in-house testing, we also support testing of "virtual" models. In this case, measurement equipment is replaced by virtual (demo) devices, which enables detection of potential design model errors.

As the last aspect of Mussa et al.'s survey [56], they outlined tool support. Developing a debugger or testing support is effort and time consuming. We can reduce these expenses using publicly available platforms. Pajunen et al. [65] shared their experience of integrating the TEMA model-based development platform with a test generator with a keyword-driven test automation open-source solution Robot framework (http://robotframework.org/). The authors claim that the amount of implementation work was rather small when compared to the benefits of the integration. The downside is the generic textual notion of the keyword, while our solution contains high-level elements defined in the model.

7.3. Summary of Related Work in Other End-User Features of MDD

During the last decades, many development aspects have been addressed thoroughly in the field of MDD. Some of the research directions are still in their infancy [66–68]. Da Silva tried to present most of the good practices in his survey [3]. Beside other findings, he found that modeling languages usually provide one or more "viewpoints". He classified viewpoints into the abstraction and perspective dimensions [3]. On the abstraction dimension, viewpoints are usually split into Computational Independent Model, Platform-Independent Model, and Platform-Specific Model. Recently, we started to investigate in this direction, since Sequencer should have a platform-independent generation of code in the future.

Challenger et al. [13] also used viewpoints, but those are connected with the perspective dimension [3]. They used MDD to develop a Multi-Agent System (MAS) to obtain autonomous and proactive properties of agents in agent-oriented software engineering. They proposed an MDD methodology, SEA_ML, that covers the whole process of analysis, modeling, code generation and implementation in the domain of MAS. The results were shown on an electronic bartering case study. Sequencer shares similarities with SEA_ML, since the whole development methodology is based on a DSML, but, as shown in this paper, we extended our MDDE with tool support (debugging, automated testing, refactoring, etc). Also, Sequencer is used in real-world problems and applied to industrial environments.

Multi-view development [13,47] is one of the features recently argued by MDD researchers. Barnett et al. [47] proposed a framework called Rapid APPlication Tool (RAPPT) for multi-view development of mobile apps, which uses a DSL for a detailed view on code and a DSML for the abstract view (e.g., page navigation). Similarly, Sequencer, from the beginning, has supported both textual [48] and modeling notation, but, the main difference is that, in the RAPPT metamodel, the individual modeling block is specified as visual, textual, or both. Sequencer is more similar to SEA_ML [13], where the model is split into several perspectives (viewpoints). Sequencer separates the main model from several "measurement" event models (OnGetData, OnTrigger, etc.). Another Sequencer viewpoint is obtained with "Custom block" [27], which splits large-scale models into smaller, more manageable models (a division of concerns), as described in Section 3.

8. Conclusions

Tool support is indispensable in any software development paradigm; the MDD should not be an exception. With this work, we would like to motivate language researchers by developing tool support for DSMLs. The construction of a DSML is only the first piece of the tool-chain needed to assist domain experts. With tools, like a debugger, we add to the domain experts' comprehension of the models. As professional programmers, domain experts also need tool support to discover the existences of software errors and locate them within a DSML model easily. The paucity of such tools can be one of the major factors that may prevent wider acceptance of MDD in the software industry.

In this paper, a debugging tool, LadyBird, is presented for the efficient detection of modeling errors. Different techniques (known to professional developers, such as execution modes, steps, breakpoints, animations, variable views, and stack traces) were applied successfully to the domain of End-User Programming. Another useful tool during MDD is automated testing. For this purpose, we extended DSML and developed SeTT, which allow the creation of test cases which can test models in combination with hardware. Another tool for End-User Programming presented in this paper is RTool, which enables restructuring of measurement procedures. All the above-mentioned tools with an underlying Sequencer are included in the DEWESoft package, which is one of the most efficient Data Acquisition systems available, used in power analysis, automotive testing, aerospace applications and many other industries.

In this paper, we also provided our experiences through 10 years of collaborating with end-users of the Sequencer. With the growing community, support was needed in the form of on-line training and webinars. Regarding support for maintenance and further extensions, continuous integration applying model-based tests to verify changes in code by an automated build allows developers to detect problems early. Additionally, we showed that building MDDE with all tool support from scratch might not be as costly as expected. Besides, our issue tracking system shows that the DSML part is at least as stable as the rest of the underlying system. To prevent further resistance of MDE within the software industry we must equip end-users with tools that work on the abstraction level of a domain. Hence, DSML compiler/interpreter must be accompanying other tools that are indispensable for end-user programming in practice, such as a debugger, testing tool and refactoring tool.

There are many development directions in such sophisticated software as Sequencer. The future work is connected mainly with the practical aspects of how to make modeling easier and more efficient for domain experts, but, even this has to be taken with a grain of salt. End-User Programming has many limitations, and new functionality needs to be introduced with caution. One possible research direction could be an investigation of the automated test generation with mutation MBT. In this perspective, we have already tested the generation of models. We used Genetic Algorithms and have some promising outcomes, but current results are not applicable for industrial settings, due to the small complexity of models and time of the new model generation.

Author Contributions: Conceptualization, T.K. (Tomaž Kos), M.M., T.K. (Tomaž Kosar); Software development, T.K. (Tomaž Kos); Demonstration, T.K. (Tomaž Kosar); Experience Report, T.K. (Tomaž Kos); Writing—original draft paper, T.K. (Tomaž Kosar); Writing—review and editing, T.K. (Tomaž Kosar), M.M.; Supervision, M.M; References, M.M.

Funding: This work was supported by the Slovenian Research Agency under Grant P2-0041.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Van Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Not.* **2000**, *35*, 26–36. [CrossRef]
2. Mernik, M.; Heering, J.; Sloane, A. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* **2005**, *37*, 316–344. [CrossRef]

3. Da Silva, A.R. Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model. *Comput. Lang. Syst. Struct.* **2015**, *43*, 139–155.
4. Ledeczi, A.; Maroti, M.; Bakay, A.; Karsai, G.; Garrett, J.; Thomason, C.; Nordstrom, G.; Sprinkle, J.; Volgyesi, P. The Generic Modeling Environment. In Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP), Budapest, Hungary, 24–25 May 2001.
5. Voelter, M.; Benz, S.; Dietrich, C.; Engelmann, B.; Helander, M.; Kats, L.C.; Visser, E.; Wachsmuth, G. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*; dslbook.org: Perth, Australia, 2013.
6. Hoyos, J.R.; García-Molina, J.; Botía, J.A.; Preuveneers, D. A model-driven approach for quality of context in pervasive systems. *Comput. Electr. Eng.* **2016**, *55*, 39–58. [CrossRef]
7. Holder, K.; Zech, A.; Ramsaier, M.; Stetter, R.; Niedermeier, H.P.; Rudolph, S.; Till, M. Model-Based Requirements Management in Gear Systems Design Based On Graph-Based Design Languages. *Appl. Sci.* **2017**, *7*, 1112. [CrossRef]
8. Kosar, T.; Mernik, M.; Carver, J. Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empir. Softw. Eng.* **2012**, *17*, 276–304. [CrossRef]
9. Kosar, T.; Gaberc, S.; Carver, J.C.; Mernik, M. Program comprehension of domain-specific and general-purpose languages: Replication of a family of experiments using integrated development environments. *Empir. Softw. Eng.* **2018**, *23*, 2734–2763. [CrossRef]
10. Wortmann, A.; Barais, O.; Combemale, B.; Wimmer, M. Modeling languages in Industry 4.0: An extended systematic mapping study. *Softw. Syst. Model.* **2019**, 1–28. [CrossRef]
11. Andrén, F.; Strasser, T.; Kastner, W. Engineering smart grids: Applying model-driven development from use case design to deployment. *Energies* **2017**, *10*, 374. [CrossRef]
12. Zanabria, C.; Andrén, F.P.; Kathan, J.; Strasser, T.I. Rapid Prototyping of Multi-Functional Battery Energy Storage System Applications. *Appl. Sci.* **2018**, *8*, 1326. [CrossRef]
13. Challenger, M.; Tezel, B.T.; Alaca, O.F.; Tekinerdogan, B.; Kardas, G. Development of Semantic Web-Enabled BDI Multi-Agent Systems Using SEA_ML: An Electronic Bartering Case Study. *Appl. Sci.* **2018**, *8*, 688. [CrossRef]
14. Coutinho, L.R.; Brandão, A.A.F.; Boissier, O.; Sichman, J.S. Towards Agent Organizations Interoperability: A Model Driven Engineering Approach. *Appl. Sci.* **2019**, *9*, 2420. [CrossRef]
15. Scaffidi, C.; Shaw, M.; Myers, B. Estimating the numbers of end users and end user programmers. In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, TX, USA, 21–24 September 2005; pp. 207–214.
16. Hutchinson, J.; Whittle, J.; Rouncefield, M. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Programm.* **2014**, *89*, 144–161. [CrossRef]
17. Freudenthal, M. Domain-Specific Languages in a Customs Information System. *IEEE Softw.* **2010**, *27*, 65–71. [CrossRef]
18. Harrison, W. From the editor: The dangers of end-user programming. *IEEE Softw.* **2004**, *21*, 5–7. [CrossRef]
19. Tezel, B.T.; Kardas, G. A Conceptual Generic Framework to Debugging in the Domain-Specific Modeling Languages for Multi-Agent Systems. In Proceedings of the 8th Symposium on Languages, Applications and Technologies (SLATE 2019), Coimbra, Portugal, 27–28 June 2019; pp. 8:1–8:13.
20. Steinberg, D.; Budinsky, F.; Paternostro, M.; Merks, E. *EMF: Eclipse Modeling Framework*, 2nd ed.; Addison-Wesley: Upper Saddle River, NJ, USA, 2008.
21. Tolvanen, J.P.; Rossi, M. MetaEdit+: Defining and using domain-specific modeling languages and code generators. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA, USA, 26–30 October 2003; pp. 92–93.
22. Bousse, E.; Degueule, T.; Vojtisek, D.; Mayerhofer, T.; Deantoni, J.; Combemale, B. Execution framework of the gemoc studio (tool demo). In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, 31 October–1 November 2016; pp. 84–89.
23. Bousse, E.; Mayerhofer, T.; Wimmer, M. Domain-Level Debugging for Compiled DSLs with the GEMOC Studio (Tool Demonstration). In Proceedings of the 1st International Workshop on Debugging in Model-Driven Engineering (MDEbug 2017), Austin, TX, USA, 17 September 2017.

24. Tolvanen, J.P.; Pohjonen, R.; Kelly, S. Advanced tooling for domain-specific modeling: MetaEdit+. In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling, Montreal, QC, Canada, 21–25 October 2007.
25. Wadler, P. Why No One Uses Functional Languages. *ACM Sigplan Not.* **1998**, *33*, 23–27. [CrossRef]
26. Gray, J.; Fisher, K.; Consel, C.; Karsai, G.; Mernik, M.; Tolvanen, J. DSLs: The good, the bad, and the ugly. In Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Nashville, TN, USA, 19–23 October 2008; pp. 791–794.
27. Kos, T.; Kosar, T.; Mernik, M. Development of Data Acquisition Systems by Using a Domain-Specific Modeling Language. *Comput. Ind.* **2012**, *63*, 181–192. [CrossRef]
28. Kosar, T.; Mernik, M.; Gray, J.; Kos, T. Debugging measurement systems using a domain-specific modeling language. *Comput. Ind.* **2014**, *65*, 622–635. [CrossRef]
29. Kos, T.; Mernik, M.; Kosar, T. Test automation of a measurement system using a domain-specific modelling language. *J. Syst. Softw.* **2016**, *111*, 74–88. [CrossRef]
30. Mannadiar, R.; Vangheluwe, H. Debugging in Domain-Specific Modeling. In Proceedings of the 3rd International Conference on Software Language Engineering (SLE), Eindhoven, The Netherlands, 12–13 October 2010; pp. 276–285.
31. Sun, Y.; Gray, J. End-User Support for Debugging Demonstration-Based Model Transformation Execution. In *Modelling Foundations and Applications*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 86–100.
32. Van Mierlo, S. A Multi-Paradigm Modelling Approach for Engineering Model Debugging Environments. Ph.D. Thesis, University of Antwerp, Antwerp, Belgium, 2018.
33. Kosar, T.; Bohra, S.; Mernik, M. A Systematic Mapping Study driven by the margin of error. *J. Syst. Softw.* **2018**, *144*, 439–449. doi:10.1016/j.jss.2018.06.078. [CrossRef]
34. Czech, G.; Moser, M.; Pichler, J. A systematic mapping study on best practices for domain-specific modeling. *Softw. Qual. J.* **2019**, 1–30. [CrossRef]
35. Associated Business Publications. 2019 Product of the Month Winners. *NASA Tech. Briefs* **2019**, *43*, 88.
36. Guo, H.F. A semantic approach for automated test oracle generation. *Comput. Lang. Syst. Struct.* **2016**, *45*, 204–219. [CrossRef]
37. Utting, M.; Pretschner, A.; Legeard, B. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **2012**, *22*, 297–312. [CrossRef]
38. Mourad, A.; Tout, H.; Talhi, C.; Otrok, H.; Yahyaoui, H. From model-driven specification to design-level set-based analysis of XACML policies. *Comput. Electr. Eng.* **2016**, *52*, 65–79. [CrossRef]
39. Marrone, S.; Rodríguez, R.J.; Nardone, R.; Flammini, F.; Vittorini, V. On synergies of cyber and physical security modelling in vulnerability assessment of railway systems. *Comput. Electr. Eng.* **2015**, *47*, 275–285. [CrossRef]
40. Planas, E.; Cabot, J.; Gómez, C. Lightweight and static verification of UML executable models. *Comput. Lang. Syst. Struct.* **2016**, *46*, 66–90. [CrossRef]
41. Erdweg, S.; Giarrusso, P.G.; Rendel, T. Language Composition Untangled. In Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12), Tallinn, Estonia, 31 March–1 April 2012; p. 7.
42. Méndez-Acuña, D.; Galindo, J.A.; Degueule, T.; Combemale, B.; Baudry, B. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Comput. Lang. Syst. Struct.* **2016**, *46*, 206–235. [CrossRef]
43. Opdyke, W.F. Refactoring Object-Oriented Frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.
44. Zhang, J.; Lin, Y.; Gray, J. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*; Springer: Berlin, Germany, 2005; pp. 199–217.
45. Hafiz, M.; Overbey, J. Refactoring Myths. *IEEE Softw.* **2015**, *32*, 39–43. [CrossRef]
46. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Boston, MA, USA, 2018.
47. Barnett, S.; Avazpour, I.; Vasa, R.; Grundy, J. Supporting multi-view development for mobile applications. *J. Comput. Lang.* **2019**, *51*, 88–96. [CrossRef]
48. Kos, T.; Kosar, T.; Knez, J.; Mernik, M. From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. *Comput. Sci. Inf. Syst.* **2011**, *8*, 361–378. [CrossRef]

49. Myers, B.A.; Ko, A.J.; Burnett, M.M. Invited Research Overview: End-user Programming. In Proceedings of the CHI '06 Extended Abstracts on Human Factors in Computing Systems, Montreal, QC, Canada, 22–27 April 2006; pp. 75–80.
50. Jannach, D.; Schmitz, T. Model-based diagnosis of spreadsheet programs: A constraint-based debugging approach. *Autom. Softw. Eng.* **2016**, *23*, 105–144. doi:10.1007/s10515-014-0141-7. [CrossRef]
51. Weintrop, D.; Wilensky, U. To block or not to block, that is the question: Students' perceptions of blocks-based programming. In Proceedings of the 14th International Conference on Interaction Design and Children, Medford, MA, USA, 21–25 June 2015; pp. 199–208.
52. Maloney, J.; Resnick, M.; Rusk, N.; Silverman, B.; Eastmond, E. The scratch programming language and environment. *ACM Trans. Comput. Educ. (TOCE)* **2010**, *10*, 16. [CrossRef]
53. Boss, A.; Stenson, C.; Ruten, J. Visual debugging technology with pencil code: Position paper. In Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), Atlanta, GA, USA, 22 October 2015; pp. 115–117.
54. Kuttal, S.K.; Sarma, A.; Burnett, M.; Rothermel, G.; Koeppe, I.; Shepherd, B. How end-user programmers debug visual web-based programs: An information foraging theory perspective. *J. Comput. Lang.* **2019**, *53*, 22–37. [CrossRef]
55. Mayer, W.; Stumptner, M. Evaluating models for model-based debugging. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, 15–19 September 2008; IEEE Computer Society: Washington, DC, USA, 2008; pp. 128–137.
56. Mussa, M.; Ouchani, S.; Al Sammane, W.; Hamou-Lhadj, A. A survey of model-driven testing techniques. In Proceedings of the 2009 Ninth International Conference on Quality Software, Washington, DC, USA, 24–25 August 2009; pp. 167–172.
57. Javed, A.Z.; Strooper, P.A.; Watson, G.N. Automated generation of test cases using model-driven architecture. In Proceedings of the Second International Workshop on Automation of Software Test, Minneapolis, MN, USA, 20–26 May 2007; IEEE Computer Society: Washington, DC, USA, 2007; p. 3.
58. Baker, P.; Jervis, C. Early UML Model Testing using TTCN-3 and the UML Testing Profile. In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques, Windsor, UK, 12–14 September 2007; pp. 47–54.
59. Bernard, E.; Bouquet, F.; Charbonnier, A.; Legeard, B.; Peureux, F.; Utting, M.; Torreborre, E. Model-based testing from UML models. In *INFORMATIK 2006–Informatik für Menschen–Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik eV (GI)*; Kollen, S.: Bonn, Germany, 2006.
60. Filho, R.S.S.; Hasling, W.M.; Budnik, C.J.; McKenna, M. Experiences using Tedeso: An extensible and interoperable model-based testing platform. *Autom. Softw. Eng.* **2013**, *20*, 299–337. [CrossRef]
61. Elallaoui, M.; Nafil, K.; Touahni, R.; Messoussi, R. Automated Model Driven Testing Using AndroMDA and UML2 Testing Profile in Scrum Process. *Procedia Comput. Sci.* **2016**, *83*, 221–228. [CrossRef]
62. Aichernig, B.K.; Auer, J.; Jöbstl, E.; Korošec, R.; Krenn, W.; Schlick, R.; Schmidt, B.V. Model-based mutation testing of an industrial measurement device. In Proceedings of the International Conference on Tests and Proofs, York, UK, 24–25 July 2014; pp. 1–19.
63. Lebeau, F.; Legeard, B.; Peureux, F.; Vernotte, A. Model-based vulnerability testing for web applications. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 18–22 March 2013; pp. 445–452.
64. Bringmann, E.; Kramer, A. Model-Based Testing of Automotive Systems. In Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, 9–11 April 2008; pp. 485–493.
65. Pajunen, T.; Takala, T.; Katara, M. Model-based testing with a general purpose keyword-driven test automation framework. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 242–251.
66. Basciani, F.; Rocco, J.D.; Ruscio, D.D.; Iovino, L.; Pierantonio, A. A tool-supported approach for assessing the quality of modeling artifacts. *J. Comput. Lang.* **2019**, *51*, 173–192. [CrossRef]

67. Babur, Ö.; Cleophas, L.; van den Brand, M. Metamodel clone detection with SAMOS. *J. Comput. Lang.* **2019**, *51*, 57–74. [CrossRef]
68. Lettner, M.; Rodas, J.; Galindo, J.A.; Benavides, D. Automated analysis of two-layered feature models with feature attributes. *J. Comput. Lang.* **2019**, *51*, 154–172. [CrossRef]