Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Test automation of a measurement system using a domain-specific modelling language

Tomaž Kos[a], Marjan Mernik[b], Tomaž Kosar[b,*]

[a] DEWESoft d.o.o., Gabrsko 11a, Trbovlje 1420, Slovenia
[b] University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, Maribor 2000, Slovenia

A R T I C L E I N F O

Article history:
Received 30 October 2014
Revised 28 August 2015
Accepted 2 September 2015
Available online 10 September 2015

Keywords:
Test automation
Domain-specific modelling languages
Usage experience

A B S T R A C T

The construction of domain-specific modelling languages (DSMLs) is only the first step within the needed toolchain. Models need to be maintained, modified or functional errors searched for. Therefore, tool support is vital for the DSML end-user's efficiency. This paper presents SeTT, a simple but very useful tool for DSML end-users, a testing framework integrated within a DSML Sequencer. This Sequencer, part of the DEWESoft data acquisition system, supports the development of model-based tests using a high-level abstraction. The tests are used during the whole data acquisition process and able to test different systems' parts. This paper shows how high-level specifications can be extended to describe a testing infrastructure for a specific DSML. In this manner, the Sequencer and SeTT were combined at the metamodel level. The contribution of the paper is to show that one can leverage on the DSML to build a testing framework with relatively little effort, by implementing assertions to it.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

The model-driven development (MDD) (Schmidt, 2006; Völter et al., 2013; da Silva, 2015) with the usage of DSMLs (Bryant et al., 2010; Kelly and Tolvanen, 2008) and their textual companion domain-specific languages (DSLs) (van Deursen et al., 2000; Fowler, 2011; Mernik et al., 2005) are becoming more and more popular (Allison et al., 2014; Prähofer et al., 2013; Saritas and Kardas, 2014; Fister Jr et al., 2011). The reason for their widespread adoption lies in the end-users' efficiencies (Selic, 2003) – within well-defined DSML/DSL programmers are more productive within a particular domain than using a general-purpose language (GPL) (Kieburtz et al., 1996; Kosar et al., 2012). This is most probably connected with abstractions within a language – DSML programming concepts are tailored according to specific domains and with that are more expressive than code written in GPL (Mernik et al., 2005; Hermans et al., 2009). Despite the repeatedly described advantages regarding industrial cases (Luoma et al., 2004; Kos et al., 2012), there are some barriers to the greater popularity of MDD and DSMLs (France and Rumpe, 2007). Often domain-specific modelling environments (DSMEs) (Kelly et al., 1996; Ledeczi et al., 2001; Steinberg et al., 2008) are not mature enough due to missing tool support (Gray et al., 2008; 2007) which leads to users' dissatisfactions with DSMLs. Lack of experience in developing programming support tools (Lindeman et al., 2011; Wu et al., 2008) and a belief in high development costs are probably the common reasons for the absence of support tools in DSMEs.

One such programming support often absent in DSMEs is automated testing infrastructure (Wu et al., 2009; Kolomvatsos et al., 2012; Kos et al., 2011b). Testing comprises up to 50% of software development time with GPLs (Myers et al., 2011) and those numbers are probably valid when developing applications with DSMLs as well. As testing takes-up precious time, we tend to carry out manual tests automatically (Fewster and Graham, 1999). In this paper we present the automated testing tool SeTT (Sequencer Testing Tool), a simple but very useful testing framework integrated within a DSML called Sequencer included within the NASA (2009 Product of the Year Winners, 2010) awarded measurement system DEWESoft (DEWESoft X2). In a previous publication (Kos et al., 2012), we showed that DSMLs are very suitable for constructing measurement systems, where physical data are captured and conversion is performed of these results into digital form (Knez et al., 2002). The Sequencer's end-users can adjust measurements and tailor the measurement procedures to their specific needs by writing models (Seidewitz, 2003) called "sequences". The measurement system DEWESoft is used within different fields such as: aviation, construction, aerospace, and mostly within the automotive industry where measurements are conducted to ensure the safety of the vehicles (Kosar et al., 2014).

* Corresponding author. Tel.: +386 22207448; fax: +386 22207272.
E-mail addresses: tomaz.kos@dewesoft.com (T. Kos), marjan.mernik@um.si (M. Mernik), tomaz.kosar@um.si (T. Kosar).

http://dx.doi.org/10.1016/j.jss.2015.09.002
0164-1212/© 2015 Elsevier Inc. All rights reserved.

The construction of the measurement procedure with Sequencer is not the last phase in the development life-cycle of the measurement system within the DEWESoft system. After the sequence has been designed, the end-users need to check for potential errors or defects, and in the case of their presence, locate and improve them. The testing of each system's module consists of comparing the acquired data with the referenced results (Kos et al., 2011b; Bringmann and Kramer, 2008). The testing of the system was manual before the development of SeTT. Manual testing was laborious and time consuming, which required concentration and precision at work. After sloppy work, errors could easily appear and could be critical for the proper functioning of the whole measurement system. In addition, testing should be repeated every time a minimal modification is made within the system (i.e., electronics, firmware, software) and this takes up additional engineering time. In order to overcome these barriers, the automated testing infrastructure SeTT was developed for Sequencer. Actually, SeTT has been developed as a language exploitation pattern (Mernik et al., 2005) extending the existing DSML Sequencer for test automation definition. Some traditional unit-testing concepts (JUnit) have been adopted to SeTT such as: test assertions, definitions of expected values, and test-reports. The domain experts can create test-cases within SeTT for each part of the measurement system, in order to check and evaluate its behaviour.

The generalised message of this paper is to show the value of the automated tests in DSMEs. The preliminary results show that automated testing of data acquisitions within the automotive domain reduced the cost of failures and increased the quality of the measurement system. Another value of the paper resides in SeTT implementation details, and lessons learned whilst developing automated testing support inside the DSML Sequencer.

The remainder of this paper is organised as follows. The related work on the DSMLs, model-based testing, and measurement systems are discussed in Section 2. Section 3 highlights the DEWESoft system and provides Sequencer details necessary for understanding the testing frameworks' implementation. The generalised advice for extending DSMLs and their use on the example of testing tool SeTT are described in Sections 4 and 5, respectively. An automated testing demonstration on a real case scenario is illustrated in Section 6. A discussion follows in Section 7. Finally, the contributions and concluding remarks are summarised in Section 8.

2. Related work

Test automation (Fewster and Graham, 1999) is a well-established field within the GPLs, where the special software controls the outcome of the application run. In the fields of the DSLs and DSMLs, work on test automation is rare. DUTF (Wu et al., 2009) is a DSL unit testing framework that can be utilised for any DSL, which is implemented with a source-to-source implementation pattern (Mernik et al., 2005; Kosar et al., 2008) with an additional constraint that a DSL construct is translated to several non-overlapping and consecutive GPL constructs. DUTF is based on the line numbers mapping process between the DSL programme and the generated GPL programme from DSL definitions. DUTF's advantage is the ability to generate unit test engines for any DSL implemented with source-to-source implementation pattern, whilst SeTT is a single DSML adjusted testing suit. Another major difference between DUTF and SeTT is in the definitions of the tests – whilst in DUTF it is necessary to define tests independently of the code under test, SeTT allows the inclusion of testing elements inside the models. Comparing with DUTF (Wu et al., 2009) reveals that SeTT needed far more implementation effort (DUTF was implemented in 3 kLOC, Sequencer in 23 kLOC, see further explanation in Section 7.3). Several reasons are connected with that, one of them is the reuse of infrastructure offered by Eclipse to integrate DUTF inside the DSME. On the other hand, integration of SeTT inside Sequencer had to be done manually. Another reason is the usage of the JUnit framework in DUTF generated GPL code, whilst in SeTT the complete test engine was implemented from scratch. On the other hand, a SeTT end-user can concentrate solely on tests and define them at the domain level. Because of the abstraction mismatch between DSLs and GPLs, in DUTF besides the DSL test, mappings for particular language need to be defined (for example, one variable in a DSL programme may not be equivalent to one variable in GPL). The amount of work needed to cover this semantic gap depends on complexity and feature-rich DSLs requiring more customisations.

More work has been done in the field of model-based testing (MBT) (Hartman and Nagin, 2004; Utting and Legeard, 2007; Filho et al., 2013). For example, Tedeso (Filho et al., 2013) is a test design studio integrated within the Eclipse platform and used by Siemens. Tedeso supports tests that are automation-based on system behaviour specifications defined by abstract models. Setting the system up for execution takes time. The complex system deployment results are a huge waste of time and many hardware errors can occur. Tedeso and Sequencer share the possibility of testing the environmental configuration and that hardware settings are automatically tested. Over a very short period of time one can test whether the system is properly defined with tests defined at the model level. However, both systems have distinctions. Tedeso uses UML-based notation and the more common way to integrate tests is by using activity diagrams and putting the code in the UML notes. On the other hand, Sequencer has its own domain-specific notation. The individual tests in Sequencer are defined by a model and use JUnit alike assert statements. Similarly, Tedeso uses its own notation to specify a test which is further during the generation phase connected with specific unit testing API (JUnit, NUnit), depending on the selected target language. On the other hand Sequencer's target programming language is Delphi, as the complete measurement system is written in this language. Tedeso also supports automatic test generation with data and control coverage guarantees. Currently, Sequencer does not support this feature, but could be part of the future SeTT extensions. Tedeso can be seen as a more extensible and interoperable solution available for GPLs, whilst SeTT is integrated within a domain-specific solution.

Interesting work on MBT in the area of measurement environment can be found by Bringmann and Kramer (2008). This paper presents a TPT tool, a Daimler Software Technology Research product. TPT is used within the field of the automotive domain. TPT tests are similar to SeTT in provided functionalities. Both systems facilitate design, execution, assessment, and test-cases report generation. Both TPT as well as SeTT, enable test portability which is reusable on different test platforms: MiL (model in the loop), SiL (software in the loop), PiL (processor in the loop), HiL (hardware in the loop), test rig (special equipment for assessing the performance of an observed part of a car) or even testing on the polygon using a real car (Bringmann and Kramer, 2008). The main difference is in the underlying infrastructure, whilst SeTT is integrated with the DSML Sequencer and measurement system DEWESoft, TPT is an independent test environment which can be integrated within any measurement system, taking into account the additional costs for configuration and the integration of different systems.

Robot Framework is an interesting generic test automation framework[1] that is meant to be used with test-driven development (TDD) (Beck, 2003) where test definition guides the developer to the final production code. On the other hand, tests in Sequencer are suitable for classical test automation where models need to be written first before being checked by tests. In our opinion, the Robot Framework is a classical domain-specific language where special keywords are used for testing data. Those high-level keywords are in some sense similar to our visual concepts defined in the model. While

1 http://robotframework.org/ .

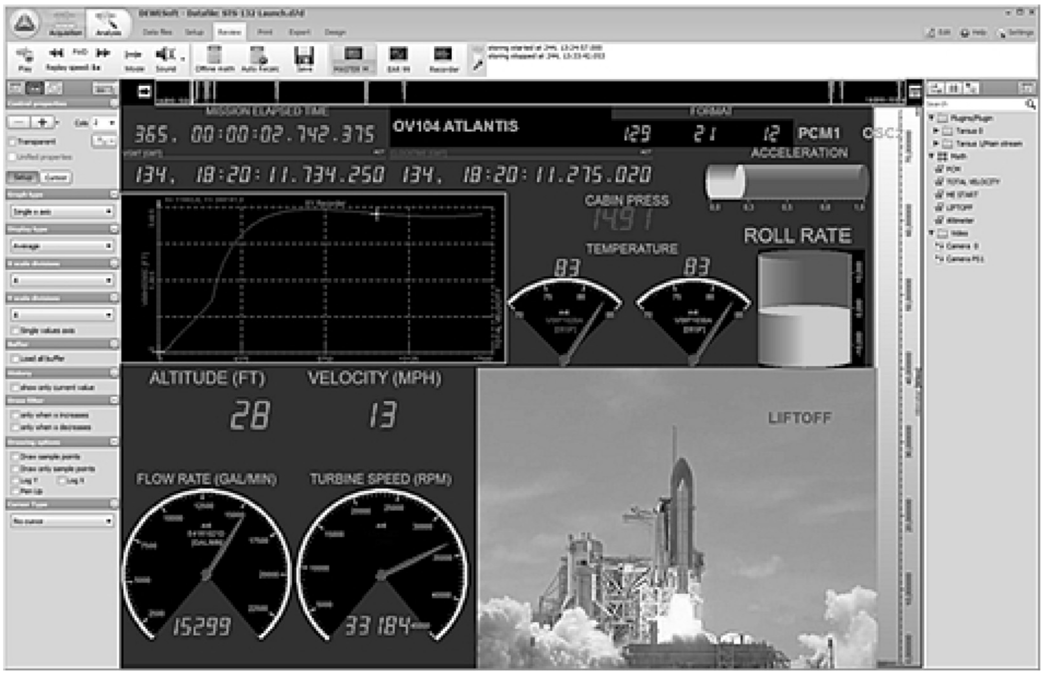


Fig. 1. Measurement system DEWESoft during space shuttle launch.

generic notion of Robot Framework targets testing of web applications, GUIs, databases, etc. our Sequencer is designed to be used for testing measurement procedures.

There exist other companies that provide commercial measurement tools. The biggest amongst them is certainly National Instruments. While DEWESoft offers Sequencer as the development environment for measurement procedures, National Instruments offers Labview[2] for defining sequences. Both measurement design platforms enable the development of measurement procedures (sequences) in a visual (graphical) manner. Sequencer contains SeTT for test automation, National Instruments provides a standalone software TestStand for that purpose. Actually, TestStand[3] is such a sophisticated test management software, with several advanced features that enable parallel test execution, schedulers, automatic test generation, integration of modules written within any test programming language, etc. SeTT is a less mature test automation tool. SeTT cannot compete with such profound systems as TestStand, our intention is to generalise our experiences and open the implementation details for other DSML developers.

The Sequencer was previously published by Kos et al. (2012), which introduced the design, implementation and usage of the DSML for a data acquisition system, whilst the Sequencer debugger was presented by Kosar et al. (2014). This current paper introduces the testing facilities in Sequencer and only those Sequencer details are given that are necessary for understanding SeTT implementation. More about the Sequencer can be found by Kos et al. (2012). We have previously published a 5 page conference paper about test automation in Sequencer (Kos et al., 2011b). That paper provided formulation of the problem, challenges and early results of the study. This earlier paper did not present implementation details nor the final results of the integrated testing facilities in Sequencer. Our current work can be seen as an extension of the earlier conference paper, containing more in-depth discussions. The contributions in this paper are the given SeTT's architecture and test automation implementation details.

[2] http://www.ni.com/labview/ .
[3] http://www.ni.com/teststand/ .

3. DEWESoft measurement system

DEWESoft is one of the leading data acquisition and analysis measurement systems used by many manufacturers within the automotive, aerospace, defence, transportation, power and other industries. DEWESoft provides synchronous data acquisition of any source from analogue, digital inputs to CAN bus, GPS, and PCM data. Another DEWESoft powerful feature is that the captured data between the measurement modules and the measurement client flows via Ethernet networks. The DEWESoft measurement system also contains analysis software that supports data storage, processing, visualisations and more. For instance, Fig. 1 shows the data acquisition during the space shuttle Atlantis launch using the DEWESoft product[4].

3.1. DEWESoft architecture

The architecture of the DEWESoft's measurement system is presented in Fig. 2. The domain experts are responsible for the measurement procedure's construction. The other users, as presented in Fig. 2, are testers and use prepared measurement procedures. They do not have the ability to modify the measurement procedure. For example, during the automotive measurements the testers are professional drivers.

The domain framework presented at the software level (Fig. 2) is in charge of the entire measurement process, as it runs the execution model and supervises the hardware. The execution model is the internal representation of a measurement procedure and directly corresponds to a model defined within the Sequencer modelling tool.

The heart of the measurement system is the hardware, which is located on the third level. At this level, the hardware components are tested such as AD/DA boards, CAN analyser, GPS receivers, telemetry boards, and others involved in the measurement system. Note that smaller defects on the circuit or problems with firmware may lead to a total measurement error. In order to control and verify the results

[4] A telemetry measurement unit (together with the software) produced by DEWESoft, is used by the aerospace industry (e.g. NASA, Raytheon, Honeywell and Lockheed Martin) and was awarded by NASA as 2009 product of the year.

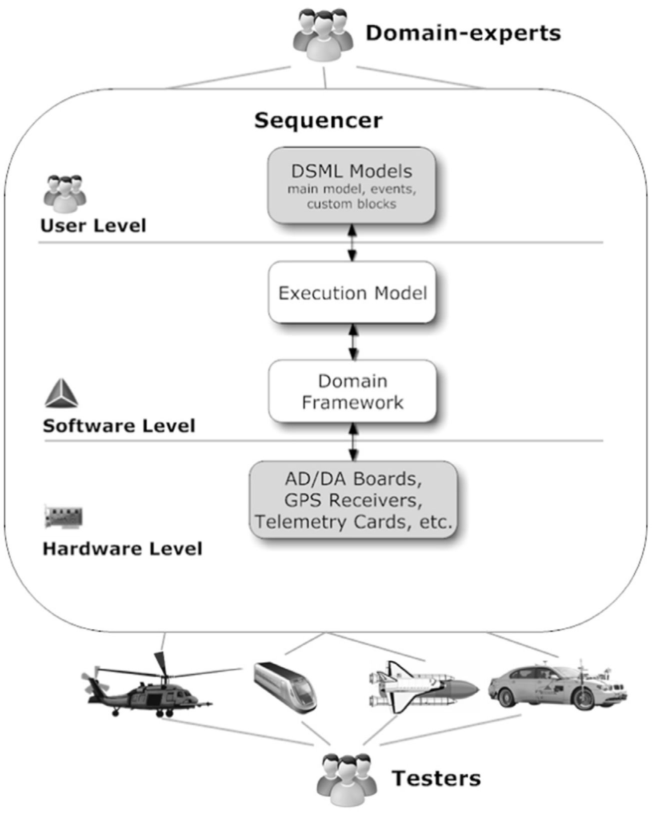


Fig. 2. Sequencer's architecture.

obtained from the hardware components, there are reference devices (calibrators, function generators, etc.) to help simulate and verify the accuracies of the obtained results (see Fig. 5).

3.2. Sequencer

In the past, measurement procedure development was becoming more and more complex within the DEWESoft measurement system. A lot of communication was needed between DEWESoft engineers and customers as customers were using DEWESoft's API and developed measurement procedures by themselves. In order to simplify the development, DEWESoft decided to build customised software. The customers were diverted from using API written in Delphi to using the Sequencer modelling tool (Kos et al., 2011a). Programmes in Sequencer can now be easily manipulated, not only by programmers but also by domain experts, thus enabling them to understand and modify data acquisition within a model.

Sequencer lets the domain expert define measurement procedures in a simple, visual manner. Fig. 3 shows the domain-expert modelling environment. In the modelling tool, building blocks (action, load setup, etc.) are visible on the left-hand side of the user-interface. Building blocks can be used in the modelling area (centre of Sequencer) using drag and drop functionality. On the right-hand side, the variable values of the selected building block can be manipulated using logical operations (variables are channels from the measurement).

The model in Fig. 3 shows a "custom model"[5] (Batory, 2006; Kulkarni and Reddy, 2003) where a simple acceleration measurement is modelled. The model starts by storing DEWESoft data file (block "StartStoring") for 60 s. After that, the measurement procedure checks if the acceleration is within the range of 1.5 g and 2.0 g (channel "Acc1"). If the condition is true, the stored data file is renamed and analysed, otherwise the stored file is deleted.

Besides the definition of the data acquisition flow, a domain expert defines the "static" part of the measurement procedure. The domain experts link custom forms, graphs, and reports to the acquired values and make them visible on the user interface during or after the measurement procedure's execution.

[5] Measurement procedure in Sequencer consists of three different model types (see Fig. 2). The first type is a main model and is mandatory in Sequencer, because it starts the measurement procedure. Events (e.g., OnTrigger) are the second model type and interrupt the main model execution. Custom models are the third model type. These models are "sub-models" and are called by other models.

3.2.1. Sequencer abstract syntax

Sequencer, as other DSMLs, is developed using a metamodel (Atkinson and Kuhne, 2003; Karsai, 1995; Hoyos et al., 2013), which defines DSML abstract syntax. The metamodel includes domain concepts that are used for modelling a concrete problem. The domain concepts correspond to actions/features from the measurement domain. The measurement concepts in Sequencer are problem independent which makes the Sequencer useful within different fields (e.g., automotive, aerospace, transportation, industrial, and civil engineering). In addition to concept definition, the notation needs to be defined within the Sequencer. Building blocks, visual representations, are assigned to constructs. They are divided into shapes and connections (relationships between concepts). In Sequencer, each shape consists of a different colour, size, and form (e.g., rectangle, diamond, and ellipse), whilst the connection is fixed and has a unique form (a line with an arrow). Each shape belongs to exactly one building block. Each building block represents an action to be executed. The execution begins in the starting building block (marked with a circle) and continues with the connected building block. More information about Sequencer can be found by Kos et al. (2012).

4. Building and integrating tools into DSME

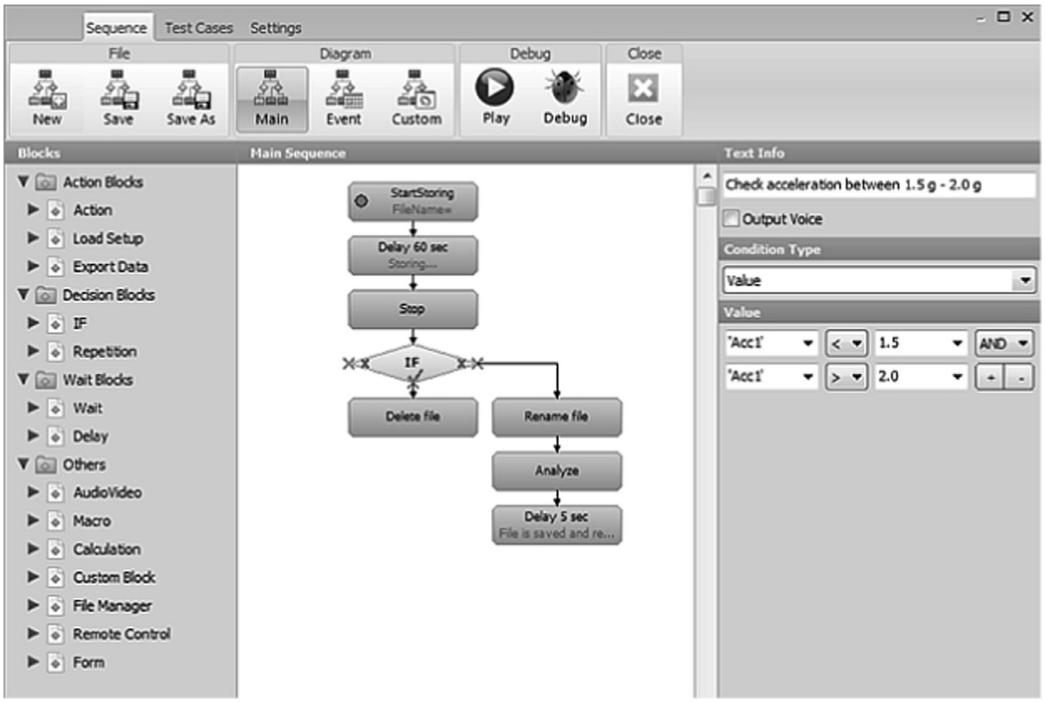
Language development tools (e.g., debuggers, test engines, and profilers) are often lacking in current DSME (Gray et al., 2008). There are different ways as to how these indispensable tools can be built and integrated within DSME. Some of the possible approaches are:

1. a poor man's approach, where tools and integration is done from scratch;
2. a tool is automatically generated from formal language specifications (e.g., Henriques et al., 2005) and manually integrated into DSME.
3. An existing GPL tool is reused for external DSMLs (e.g., Wu et al., 2008, 2009) or internal DSMLs (e.g., Renggli et al., 2010)
4. DSML is extended in invasive or non-invasive manner to support a particular tool.

Since the 2nd option requires DSML formal semantic description, which is still challenging (Bryant et al., 2011) we opted for non-invasive extension of DSML, which corresponds to language extension as a special case of language composition (Erdweg et al., 2012; Mernik, 2013).

This section is dedicated towards methodology which helped us to incorporate SeTT into Sequencer. This step-by-step procedure for the DSML extension will be used in the next subsection where we explain how authors applied this methodology to SeTT.

For the development of the base DSML we have used a classical DSML development methodology with the following phases: domain analysis, design, and implementation (see left-hand side of Fig. 4). Same phases have been also applied for the development of DSML extension as indicated on the right-hand side of Fig. 4. All of the extension phases use as an input an output of the corresponding phases from the development of the base language (represented with Δ between base and extended DSML) and output of the previous phase



Check acceleration between 1.5 g - 2.0 g
Output Voice
Condition Type: Value
'Acc1' < 1.5 AND
'Acc1' > 2.0

Fig. 3. Sequencer modelling environment.

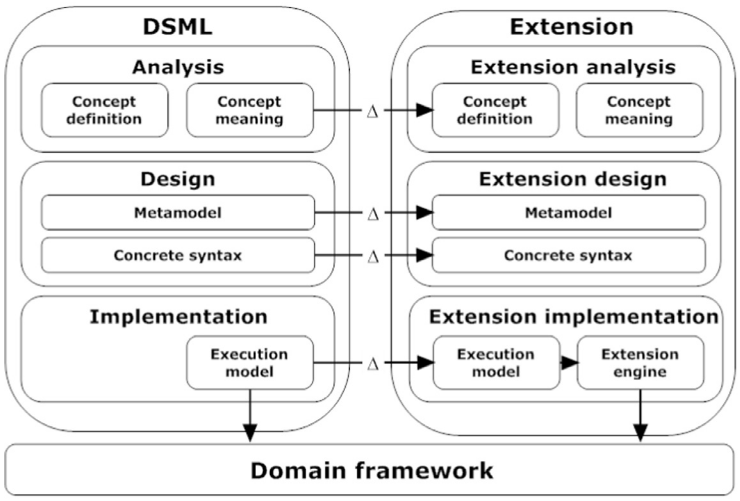


Fig. 4. Generalised overview of methodology used for Sequencer extension with SeTT.

in DSML extension methodology. The first phase in our methodology is connected with extension analysis. This phase is divided into two steps: firstly, we have to find those domain concepts that should be included in the existing DSML and secondly, we have to find the meaning for each concept.

The next step is connected with extension design. As already mentioned a DSML can be extended in invasive or non-invasive manner. In the first case, implementing a language extension involves changing the implementation of the base language. In this case the base DSML is changed, which was unacceptable from the point of view of already written measurement procedures. In the case of non-invasive language extension, base language implementation must be used unchanged (Erdweg et al., 2012). Non-invasive DSML extension is possible for DSML testing support where new concepts are orthogonal to existing DSML concepts. Therefore, two DSMLs were derived from the unique metamodel and used in the same domain-specific modelling environment for sequences and test-cases separately (see the top bar in Fig. 3). In this way, the base DSML uses the "original" metamodel, while the second DSML uses the metamodel that includes concepts for the DSML support tool (i.e. testing framework). Next step in the extension design is to define the concrete syntax of the new building blocks (see Fig. 4). Of course, the building block appearance has to follow the example of the building blocks from the base concrete syntax.

Regarding the implementation of the extension, the language transformation of the base DSML can be completely reused. As our extension concepts are orthogonal to base DSML constructs, the semantics of the base DSML is without modification (the execution model calls the same functionalities from the domain framework). On the other hand, transformations for the new metamodel constructs need to be written from scratch. For this purpose an extension engine can be written. As will be shown in the following section, the domain test engine (Fig. 5) has been written separately from the base DSML and therefore can be reused as a testing framework for other DSMLs.

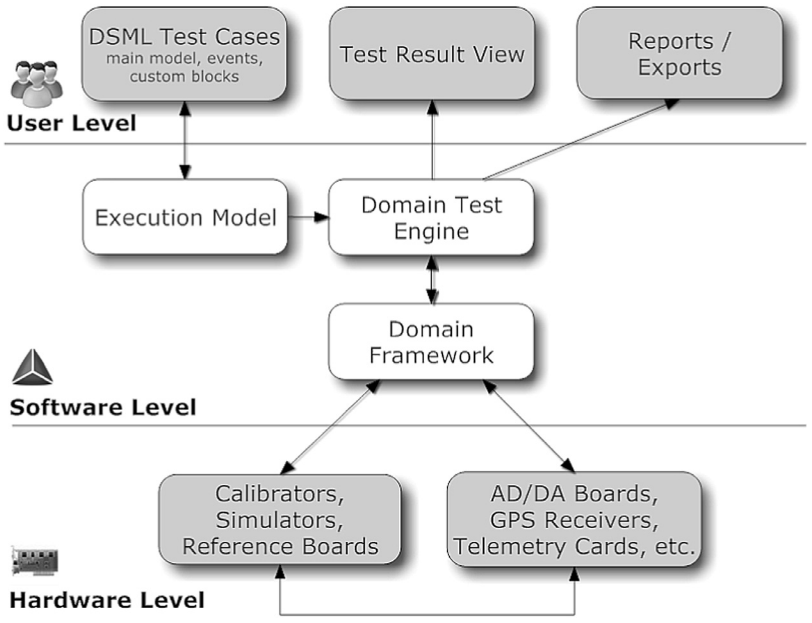


Fig. 5. DSML testing tool.

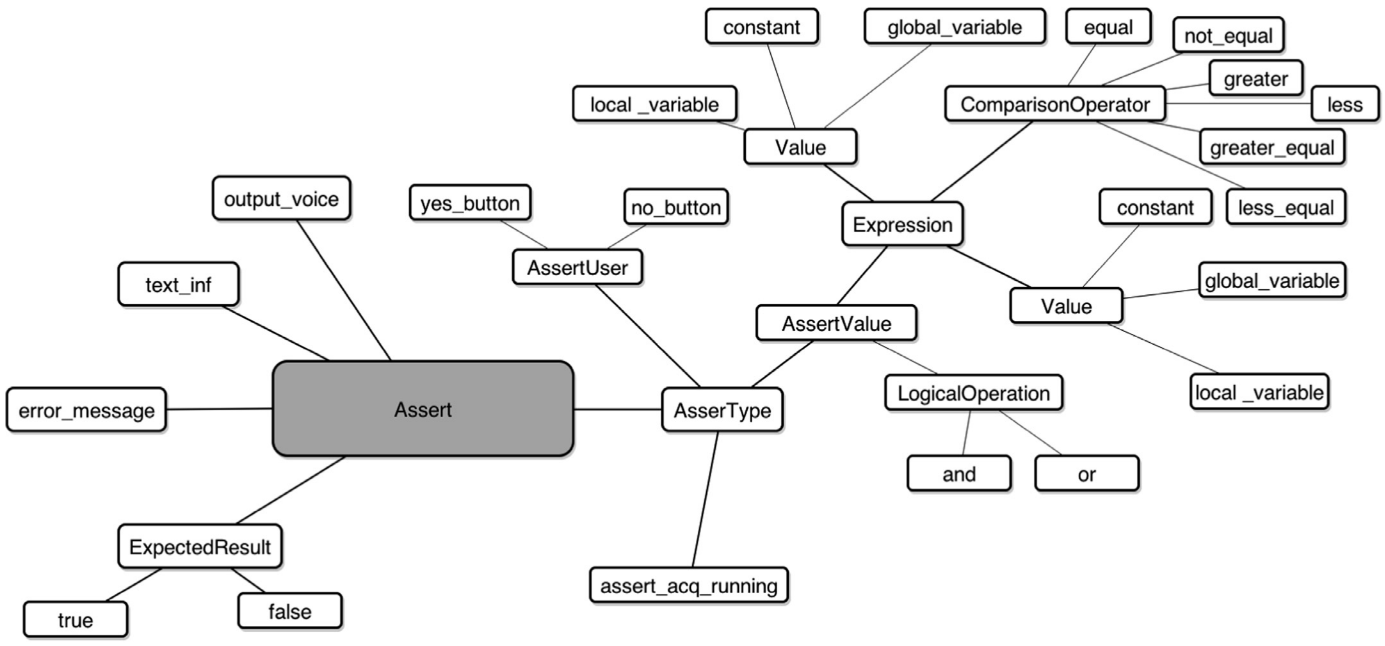


Fig. 6. Hierarchical decomposition of new concept Assert.

In contrast, the test-case result view is hard-coded in the DSME and must be implemented for each DSME separately.

5. SeTT – Sequencer's extension

In general, our methodology is divided into three phases: extension analysis, extension design and extension implementation. In the following section we will use it on the case study of SeTT.

5.1. Extension analysis

In the next subsections, we present our own advice on how to analyse the DSML extension. As the outcome of this phase are concepts' definitions which represent DSML extension.

5.1.1. Concept definition

During the "Concept Definition" step in the "Extension Analysis" phase (see Fig. 4, again) we identified the concept "Assert" as the only domain concept that needs to be integrated within existing DSML. The purpose of this construct is to validate specific systems' outcomes. Hierarchical decomposition of "Assert" is presented in Fig. 6. The root feature "Assert" contains text information (text_inf), output voice (output_voice), type of assertion (AssertType), expected result (ExpectedResult), and error message (error_message). Note, that the features starting with an uppercase letter (AssertType, ExpectedResults) are further decomposed.

5.1.2. Concept meaning definition

The next step within "Extension Analysis" of our methodology (see Section 4, again) was to define the meaning of extension

Table 1
Feature definitions for the concept “Assert”.

Feature	Type	Attribute type	User defined
text_inf	Attribute	Text	Yes
output_voice	Attribute	Boolean	Yes
AssertType	Attribute	Enum	Yes
ExpectedResult	Attribute	Boolean	Yes
error_message	Attribute	Text	Yes

features (Fig. 6). In Table 1 we defined the meaning for the main feature. As stated above, the only feature that represented a concept was “Assert”, the rest of the features were assigned as attributes to the concept “Assert”. Furthermore, the attributes were observed and their types assigned to them. For instance, “text_inf” contains the test name and clearly must be of the “text” type. Information interesting for further development is designated according to whether this information is user defined or calculated during the measurement procedure. For instance, attribute “ExpectedResult” is of boolean type and the user defines whether positive (true) or negative (false) values can be expected from the measurement procedure execution.

5.2. Extension design

The next step in our methodology is to design the syntax of the extension – metamodel and concrete syntax (see Fig. 4, again).

5.2.1. Extension of metamodel

Fig. 7 presents the extended metamodel – the Sequencer metamodel together with extension for test automation. First, let us introduce the Δ metamodel, the base DSML metamodel, from our methodology (see Fig. 4, again). The metamodel in GME (Ledeczi et al., 2001) is defined as a class diagram using UML notation. For the clarity of Fig. 7 we left out the attributes of classes. The root class in the metamodel is “TSequence”. This class contains instances of “TConnection” and “TSeqItem” classes, with “zero to many” relation. Basically, this means that the model can contain connections and items. Classes that are coloured grey are the concepts that have visual representation as building blocks in the measurement procedure. Such an example is “TConnection”. On the other hand “TSeqItem” is white and does not have any visual representation within a model. Instead, classes that are inherited from the base class “TSeqItem” have visual representation (“TAction”, “TFileManager”, etc.). Furthermore, some classes have common characteristics and are therefore combined into the “TBaseAction”, “TPause” and “TControl” classes within our metamodel.

New classes, “TTest” and “TAssert”, as marked in Fig. 7, extend Δ metamodel (Sequencer metamodel). Both classes are inherited from the “TSeqItem” bring the possibility of defining the automated tests in the Sequencer. The first class, “TTest”, follows the practice of classes “TBaseAction”, “TPause” and “TControl”, to have a class that joins the common characteristics of classes. In this way, a new test can be inserted as a “TTest” subclass. “TAssert” corresponds to the concept

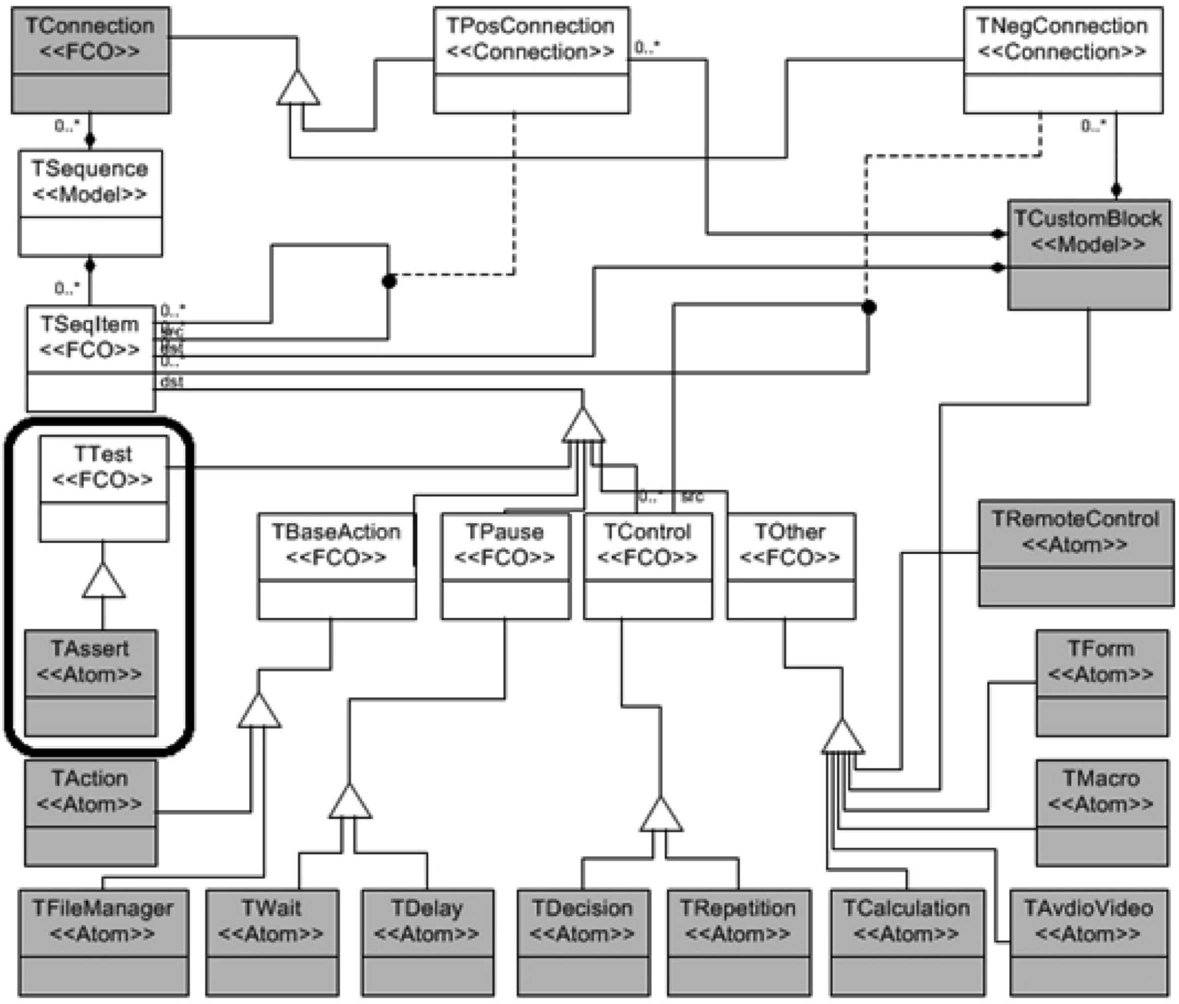


Fig. 7. Introduction of the test automation to Sequencer metamodel.

"Assert" identified within domain analysis and contains the attributes defined in Table 1.

5.2.2. *Concrete syntax of extension*

The next step after the metamodel extension is to establish the concrete syntax of the new classes. In the case of the class "Assert", the shape with colour, size, and form (rounded rectangle) was chosen. The class appearance follows the example of other classes, only the colour in the class "Assert" is different.

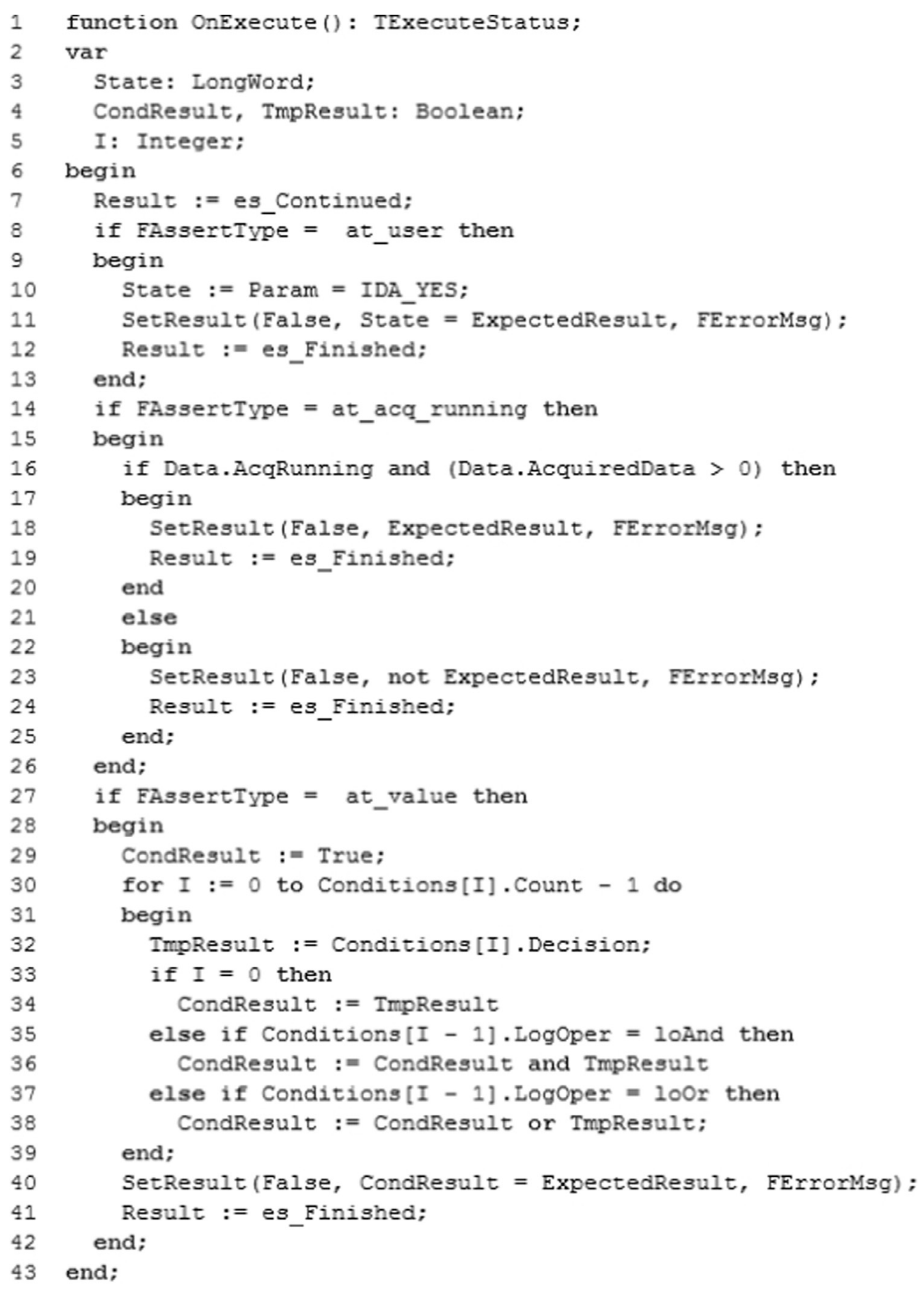
5.3. *Extension implementation*

The most difficult part of any DSML extension methodology is to handle the implementation of the DSML extension (see Fig. 4, again). Automated generation of, for instance, testing frameworks is still far away as there is no standard way of describing the semantics of a DSML (Bryant et al., 2011) or its extension.

In this section we present how our testing framework extension has to be implemented and connected with the implementation of the DSML.

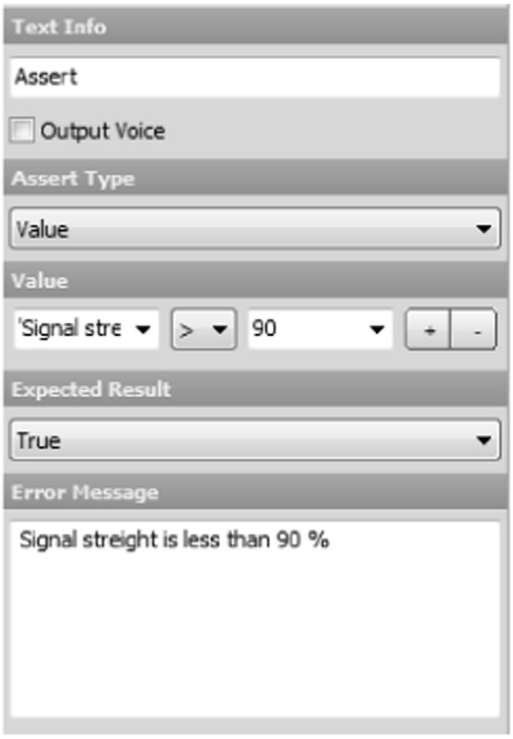
5.3.1. *Execution model*

As described in Kosar et al. (2014), each language construct is transformed to the execution block as a class with three main functionalities defined in Object Pascal: "start" (starts the block execution), "stop" (stops the block execution) and "execute" (contains the main functionality of the execution block). In the case of construct "Assert", functionality "execute" is transformed into the code shown in Fig. 8. Function OnExecute() returns the result of a test assertion in the second parameter of a method SetResult (lines 11, 18, 23, and



```
1   function OnExecute(): TExecuteStatus;
2   var
3     State: LongWord;
4     CondResult, TmpResult: Boolean;
5     I: Integer;
6   begin
7     Result := es_Continued;
8     if FAssertType =  at_user then
9     begin
10      State := Param = IDA_YES;
11      SetResult(False, State = ExpectedResult, FErrorMsg);
12      Result := es_Finished;
13    end;
14    if FAssertType = at_acq_running then
15    begin
16      if Data.AcqRunning and (Data.AcquiredData > 0) then
17      begin
18        SetResult(False, ExpectedResult, FErrorMsg);
19        Result := es_Finished;
20      end
21      else
22      begin
23        SetResult(False, not ExpectedResult, FErrorMsg);
24        Result := es_Finished;
25      end;
26    end;
27    if FAssertType =  at_value then
28    begin
29      CondResult := True;
30      for I := 0 to Conditions[I].Count - 1 do
31      begin
32        TmpResult := Conditions[I].Decision;
33        if I = 0 then
34          CondResult := TmpResult
35        else if Conditions[I - 1].LogOper = loAnd then
36          CondResult := CondResult and TmpResult
37        else if Conditions[I - 1].LogOper = loOr then
38          CondResult := CondResult or TmpResult;
39      end;
40      SetResult(False, CondResult = ExpectedResult, FErrorMsg);
41      Result := es_Finished;
42    end;
43  end;
```

Fig. 8. Implementation of function OnExecute for metamodel's component TAssert.


Text Info

Assert

Output Voice

Assert Type

Value

Value

'Signal stre > 90

Expected Result

True

Error Message

Signal streight is less than 90 %

Fig. 9. Definition of "at_value" assertion type.

40) and can be paralleled with JUnit's assertEquals function where the actual result and the expected result from acquisition are compared. First, within the function OnExecute(), the assertion type is checked. In SeTT three different assertions are possible. If assert type is "at_user" (line 8), the function SetResult returns the result by comparing user's input with the expected result (line 11). If the assertion type is "at_acq_running" (line 16), the SetResult function returns the acquisition status (lines 18 and 23). If assertion type is "at_value", the assertion checks the values from the acquisition. An example of the last assertion type is shown in Fig. 9. Here, the end-user is checking if the signal strength from data acquisition is good enough. An actual value of assertion is defined with signal strength that should be greater than 90. The expected value must be valid (see region "Expected Result" in Fig. 9).

More precisely, the last assertion type "at_value" is a composition of logical expressions. The code in Fig. 8 loops through variable Conditions containing individual logical expressions (lines 30–39). The result of the first logical expression is assigned to the entire condition result (lines 33 and 34, variable CondResult). All other logical expressions are first compared with the logical operator loAnd and loOr and then combined with previous logical expressions (lines 35–38) for the condition result.

5.3.2. *Domain test engine*

Unit testing is usually connected with the JUnit testing framework (JUnit). JUnit is a Regression Testing Framework used by developers when implementing unit tests in Java. The JUnit Framework allows developers to easily integrate their tests with the code under test, running the tests and monitoring the results. SeTT and JUnit framework share the common ideas of unit testing: definition of tests with test assertions, definitions of expected values, etc. Additionally, the JUnit framework supports "suits", test aggregation, which is inapplicable in the Sequencer. JUnit framework or similar solutions (DUnit for Delphi users) could also be used in Sequencer – test-cases could be written after the generated code. However, these would be feasible only for programming engineers, not for domain engineers without programming skills. Our intention was to support Sequencer end-users and to create test-cases with DSML instead of GPL (to provide all DSML benefits within the same tool). The purpose of SeTT's domain test engine is equivalent to the JUnit framework. Test-cases defined in SeTT test different measurement procedures and/or their parts. The domain test engine transforms those test-cases into execution models and processes them in the same order as the test-cases were defined by the end-user. The test-case in turn is sent to the domain framework for execution. The domain test engine maintains a status for each test-case and returns the results back to the user level when all execution models have been completed.

Test-cases can have one of the following statuses: Passed, Failed, Error, Not Executed, and In Progress. Each test-case starts with the status "In Progress". The test remains in this state until the test produces a result (pass, fail, or error). If a test-case does not have an assert building block, the test always produces a Pass result. In addition to the execution status, the domain test engine receives an error message for the test-case (if necessary).

The domain test engine class diagram is presented in Fig. 10. The root class is a TTestEngine which contains a start time (attribute StartTime) and a duration of all executions (attribute Duration). Actually, the class TTestEngine contains a reference to the test-cases (TestCases) and a reference to the current executing test-case (CurrentTestCase). The class TTestCase holds an execution model start time (attribute StartTime), an execution duration (attribute Duration), a reference to the execution model (attribute ExecuteModel), and an error message (attribute ErrorMsg). Note, that the execution model is of type TSequence, which is the main class of our metamodel (see Fig. 7, again). Additionally, the class TTestCase contains the test result (attribute TestResult) and the current test result (attribute CurrTestResult). The attribute TestResult holds the status "InProgress" during the whole test-case execution and receives the final result at the end from the attribute CurrTestResult. On the other hand, the attribute CurrTestResult value changes with "Assert" execution.

The execution starts by calling the method StartTesting(), and ends with method StopTesting() within class TTestEngine. The first method initialises the test-cases (sets attributes TestResult and CurrTestResult to status trNotExecuted), sets the execution start time

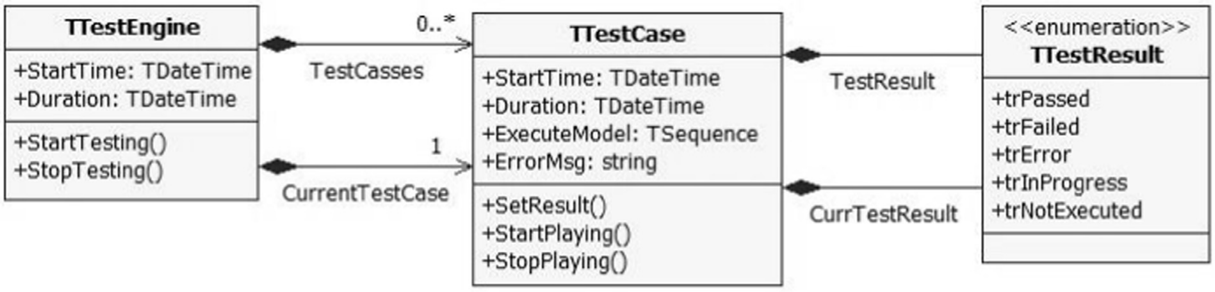
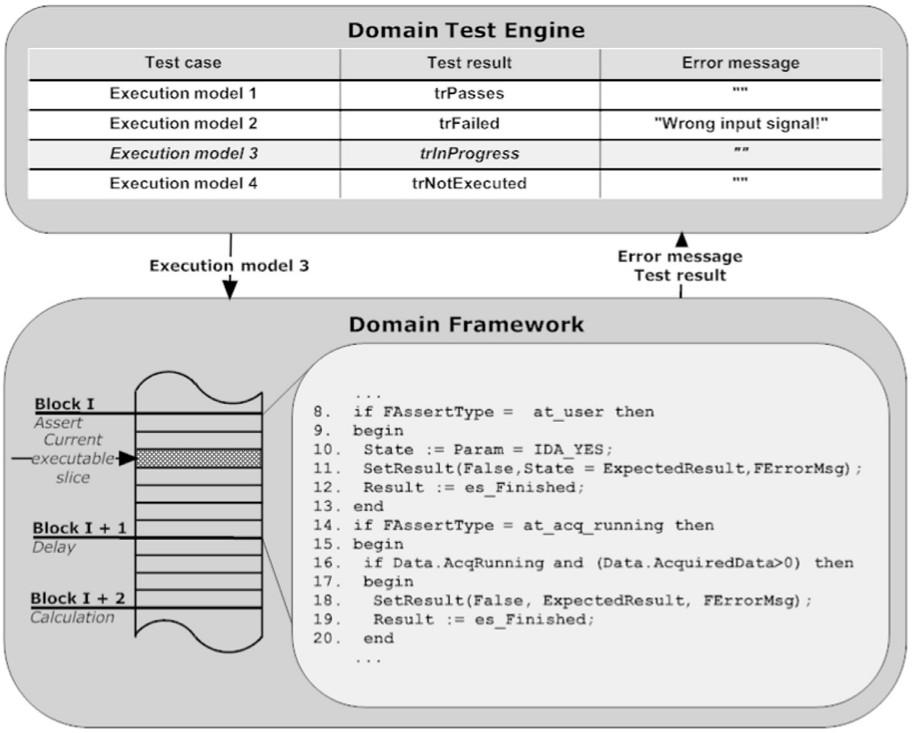


Fig. 10. Domain test engine class diagram.

```
procedure TTestCase.SetResult(IsError, PassedResult: Boolean; ErrorMessage:string);
begin
  if IsError then
  begin
    CurrTestResult := trError;
    ErrorMsg := ErrorMessage;
  end
  else if PassedResult and (TestCase.CurrTestResult = trInProgress) then
  begin
    TestCase.CurrTestResult := trPassed;
  end
  else if not PassedResult and not (TestCase.CurrTestResult in [trFailed, trError]) then
  begin
    CurrTestResult := trFailed;
    ErrorMsg := ErrorMessage;
  end;
end;
```

Fig. 11. Function SetResult in class TTestCase.



Test case	Test result	Error message
Execution model 1	trPasses	""
Execution model 2	trFailed	"Wrong input signal!"
Execution model 3	*trInProgress*	""
Execution model 4	trNotExecuted	""

```
...
if FAssertType =  at_user then
begin
 State := Param = IDA_YES;
 SetResult(False,State = ExpectedResult,FErrorMsg);
 Result := es_Finished;
end
if FAssertType = at_acq_running then
begin
 if Data.AcqRunning and (Data.AcquiredData>0) then
 begin
  SetResult(False, ExpectedResult, FErrorMsg);
  Result := es_Finished;
 end
...
```

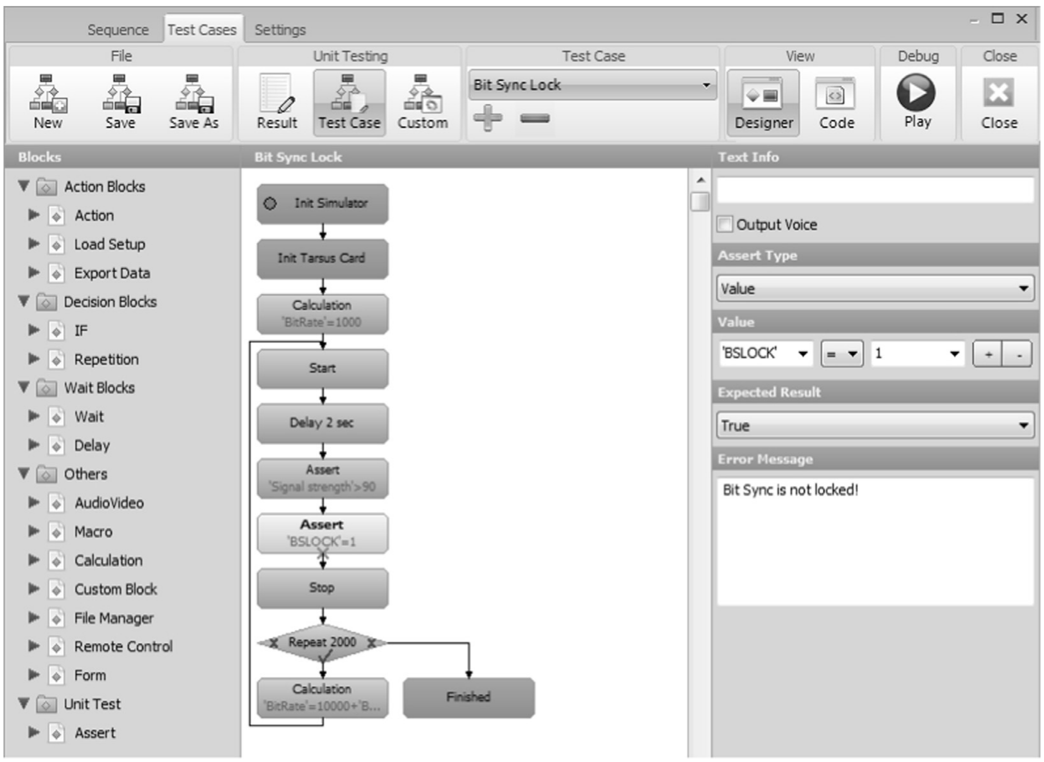
Fig. 12. Test-cases execution.

(attribute StartTime) and prepares the first test-case for execution (sets attribute CurrTestCase to first test-case from the TestCases list). Each test-case is called with the method StartPlaying(), which starts the execution model. Moreover, attribute StartTime is set to the current system time, attribute TestResult to the state trInProgress. During the model execution, domain framework returns test results through method SetResult (Fig. 11). This method sets current test result (attribute CurrTestResult) and information about error (attribute ErrorMsg). The first parameter of the method SetResult contains information as to whether there was any error during the execution, second parameter returns information about executed assert statement. When the function returns a failure status, also an error message (parameter ErrorMessage) is set. At the end of the test-case, method StopPlaying() is called, which returns the test result (attribute CurrTestResult).

An example of the test execution within the domain test engine is shown in Fig. 12. The domain test engine consists of four test-cases. The first test-case (Execution model 1) was completed successfully (status trPassed). The second test-case (Execution model 2) produced an error (status trFailed) with associated error information "Wrong input signal!". The third test-case (Execution model 3) is currently running (status trInProgress) and the fourth (Execution model 4) is waiting for the execution (status rhNotExecuted). A source code of the function OnExecute() is waiting to be executed (complete function is shown in Fig. 8).

6. An example

Nowadays, it is hard to imagine large and complex systems, such as satellites, rockets and spacecrafts without the use of PCM



Text Info

Output Voice

Assert Type: Value

Value: 'BSLOCK' = 1

Expected Result: True

Error Message: Bit Sync is not locked!

Fig. 13. Creating a test-case within a modelling tool.

telemetry. Such communication enables the automated monitoring, saving, and alerting of the measured data necessary for a safe and efficient device operation. Space agencies such as NASA, ESA, and other agencies, use telemetry to acquire data from spacecraft and satellites. Thousands of telemetry data are transmitted via the PCM signal from a spacecraft to the earth-system, where the data are captured, decoded, displayed and subsequently analysed. Such systems must be of high quality, and therefore need to be tested to perfection. In order to demonstrate the SeTT, the testing tool was used on PCM telemetry.

6.1. Test-case definition

Electronic systems often contain complex devices, circuits, programmes and interfaces. These must all work together within a variety of conditions. The telemetry acquisition station is no exception. The next example uses the DEWESoft together with TarsusHS-PCI-01 Processor Board.[6] As a test-case on this device, a bit-synchronizer was performed. A simulator (PCM transmitter) is needed, in order to verify different bit-rates of the PCM bit-synchronizer. The simulator sends a PCM signal onto which our tested system is able to lock. During the testing, the transmitter and the receiver bit rates (from 1 kbps to 20 Mbps) are changed and verified. On this basis, the decision is made whether the test-case has succeeded or not.

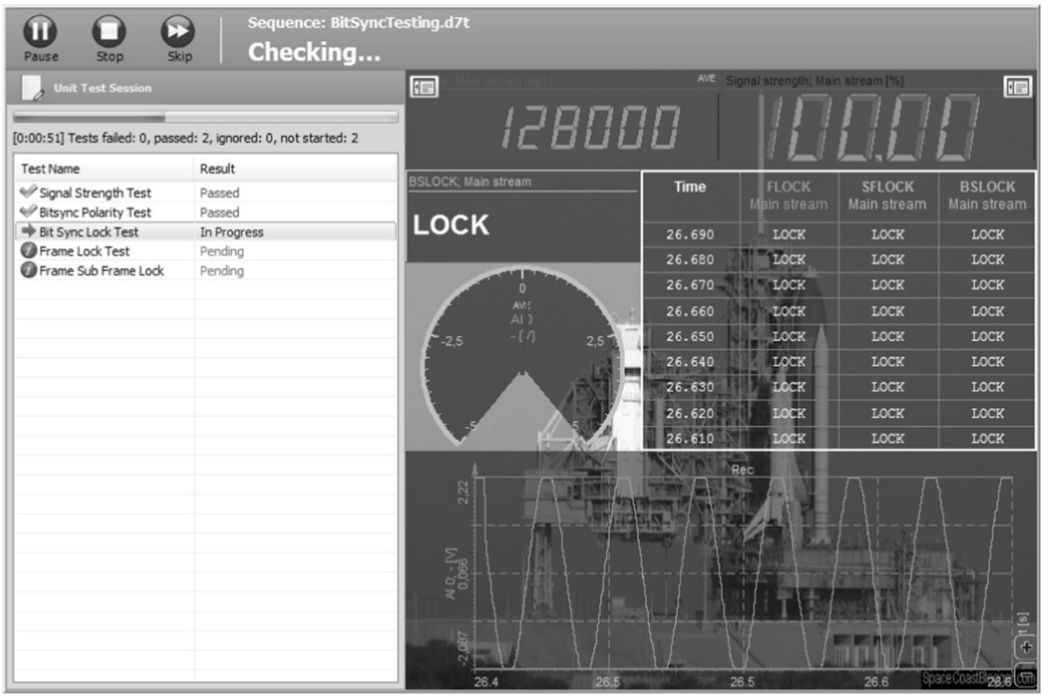
The bit-synchronizer within the Sequencer is shown in Fig. 13, with a test-case named “Bit-sync Locking”. In the “Bit-sync Locking” test-case, the procedure begins by calling two functions (blocks “Init Simulator” and “Init Tarsus Card”), which initialises the simulator and the testing board where some settings are defined such as PCM code type, signal polarity, and others (the starting block is marked with a circle).

The measurement procedure continues with a stream bit rate definition (attribute BitRate set at 1000 in a building block “Calculation”). Then the data acquisition is started (block “Start”). The purpose of the building block “Delay” in the sequence is to stabilise the signal. After 2 s delay, the first “assert” block validates the signal strength (signal-strength must be higher than 90%, otherwise the test-case fails). The second assertion determines whether the bit-sync status is locked, In this case the attribute “BSLOCK” must be 1, see attribute definition in Fig. 13. Note, that the Sequencer right-hand side contains properties that can be configured for the selected building block. As the “Assert” block was selected in the test-case, assert properties such as the expected result, error-message, etc. can be defined. At the “Stop” block, the data acquisition is stopped. In the block “Repeat 2000” the value of attribute BitRate is tested (note that data acquisition needs to be repeated 2000 times). In block “Calculation” each repetition adds to attribute BitRate 10 kbps (10 kbps $*$ 2000 = 20 Mbps). After 2000 repetitions the test-case is completed (block “Finished”) and results can be observed.

6.2. Test-case result view

During the test-case execution, the user can observe the test-case results and acquired values. Following the example of unit testing, SeTT supports running several test-cases within one test execution (Zhu et al., 1997). Besides the “Bit-sync Locking” test-case, Fig. 14 shows other test-cases executions: Signal Strength, BitSync Polarity, Frame Lock and Frame Lock Test. Those tests are very common for testing a telemetry acquisition station on a space shuttle and are usually tested together. On the left-hand side of the graphical user interface in Fig. 14, we can observe the elapsed time, number of failed tests (none), tests passed (two passes), ignored tests (zero ignored tests), and tests not yet started (two). The test-cases list underneath the header-view, indicates current test-cases results. During the test execution, the acquired data can be observed on the right-hand side of Fig. 14. For the current test-case, the input signal can be monitored together with the bit-rate, and bit-status, which are used for the validation.

[6] Bit-synchronizer processor board developed by Ulyssix Technologies, Inc.



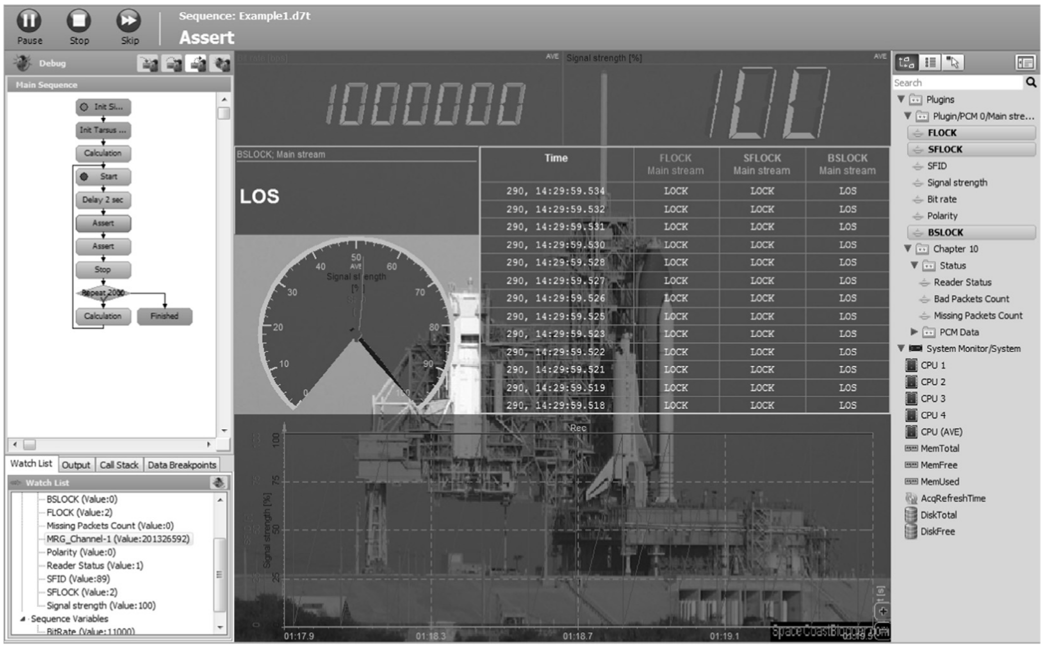
Sequence: BitSyncTesting.d7t
Checking...

[0:00:51] Tests failed: 0, passed: 2, ignored: 0, not started: 2

Test Name	Result
Signal Strength Test	Passed
Bitsync Polarity Test	Passed
Bit Sync Lock Test	In Progress
Frame Lock Test	Pending
Frame Sub Frame Lock	Pending

Time	FLOCK Main stream	SFLOCK Main stream	BSLOCK Main stream
26.690	LOCK	LOCK	LOCK
26.680	LOCK	LOCK	LOCK
26.670	LOCK	LOCK	LOCK
26.660	LOCK	LOCK	LOCK
26.650	LOCK	LOCK	LOCK
26.640	LOCK	LOCK	LOCK
26.630	LOCK	LOCK	LOCK
26.620	LOCK	LOCK	LOCK
26.610	LOCK	LOCK	LOCK

Fig. 14. Test-cases execution.



Sequence: Example1.d7t
Assert

Time	FLOCK Main stream	SFLOCK Main stream	BSLOCK Main stream
290, 14:29:59.534	LOCK	LOCK	LOS
290, 14:29:59.532	LOCK	LOCK	LOS
290, 14:29:59.531	LOCK	LOCK	LOS
290, 14:29:59.530	LOCK	LOCK	LOS
290, 14:29:59.528	LOCK	LOCK	LOS
290, 14:29:59.527	LOCK	LOCK	LOS
290, 14:29:59.526	LOCK	LOCK	LOS
290, 14:29:59.525	LOCK	LOCK	LOS
290, 14:29:59.523	LOCK	LOCK	LOS
290, 14:29:59.522	LOCK	LOCK	LOS
290, 14:29:59.521	LOCK	LOCK	LOS
290, 14:29:59.519	LOCK	LOCK	LOS
290, 14:29:59.518	LOCK	LOCK	LOS

Watch List
- BSLOCK (Value:0)
- FLOCK (Value:2)
- Missing Packets Count (Value:0)
- MRG_Channel-1 (Value:201326592)
- Polarity (Value:0)
- Reader Status (Value:1)
- SFID (Value:89)
- SFLOCK (Value:2)
- Signal strength (Value:100)
- Sequence Variables
  - BitRate (Value:11000)

Fig. 15. Debugging test-case in Sequencer.

6.3. *Debugging of failed test-cases*

The test-cases for the telemetry system have to follow a common pattern in the Sequencer. First, we set all the necessary parameters (sensors, bitsync, framesync, and decom). The procedure continues with a space shuttle data insertion. The end-user has to enter these data at the beginning of each measurement. This information is used to calculate values. When the measurements are finished, the engineers can analyse the acquired data. Besides, additional features are available in Sequencer such as creating standard reports, file exports, etc. The desired functionality must also be tested with SeTT. In this way, the software testing covers the software accuracy (calculating values, etc).

When we tested a new PCM module together with the current software version DEWESoft X2, the “Bit Sync Lock” test-case failed. In such a case, it is common to use a debugging tool such as the DSML debugger LadyBird (Kosar et al., 2014) included in Sequencer and apply the following procedure for the detection of the bug. We used the source breakpoint to stop the sequence on the “Start” block in Fig. 15. After that, we executed a step-by-step sequence and monitored

```
unit BitSyncLockTest;

interface

uses
  TestFrameWork, DEWEsoft_TLB;

type
  TBitSyncTest = class(TTestCase)
  private
    DeweApp : App;
    function GetChannelByName(ChName : string) : IChannel;
    function GetDWTime : double;
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestBitSyncLock;
  end;

implementation

{ TBitSyncTest }

procedure TBitSyncTest.TestBitSyncLock;
var
  I : Integer;
  Channel : IChannel;
  Value : Double;
begin
  InitSimulator;
  initTarusCard;

  Channel := GetChannelByName('BitRate');
  Channel.AddAsyncIntegerSample(1000, GetDWTime);

  for I := 0 to 1999 do
  begin
    DeweApp.Start;

    Sleep(2000);

    Channel := GetChannelByName('Signal strength');
    Value := Channel.GetValueAtAbsPos(-1);
    CheckTrue(Value > 90, 'Signal strength < 90%!');

    Channel := GetChannelByName('BSLOCK');
    Value := Channel.GetValueAtAbsPos(-1);
    CheckEquals(1, Value, 'BitSync is not locked!');

    DeweApp.Stop;

    Channel := GetChannelByName('BitRate');
    Value := Channel.GetValueAtAbsPos(-1);
    Channel.AddAsyncIntegerSample(Value + 10000, GetDWTime);
  end;

  Finished;
end;

...
```

Fig. 16. Creating a test-case outside DSME.

variables. Note, that no debugging action is needed on Delphi generated code, since all the debugging is done on a DSML level. During debugging, we discovered that the bit rate of the PCM signal was set incorrectly because the measured bit rate was the same even if we changed the bitrate of the output PCM signal.

7. Discussion

In this section, the benefits of developing the DSML testing tool are discussed. Then, SeTT development effort is summarised.

7.1. Advantages of test automation within measurement systems

Since the year 2000, more than one thousand measurement systems have been developed using the DEWESoft package. In order to increase the software quality, reduce the development time and consequently a faster time to market, the development team needs to have a flexible, easy to use, and an effective testing environment. Over these years, the advantages of having the testing tool have been identified and the production of SeTT was inevitable. The objective of system testing is to isolate a part of the system and validate its accuracy. Test automation can achieve this and enable other benefits, such as testing without human influence (elimination of the human errors), test repetitions, and reusability. In addition to these benefits, automated tests are significantly faster than the manual tests performed by humans. Our experiment confirmed that. In Table 2, eight different measurement procedures for PCM telemetry were tested using automated testing tool and compared with manual tests. It can be observed that manual testing in Test 8 took approximately 8 h and 39 min, whilst the automated testing took only one hour and 43 min. The big difference in time is caused by multiple repetitions of the manual configurations within the PCM simulator as well as the PCM receiver.

Another advantage of test automation is that the costs of errors during measurement procedures can be significantly reduced, if the tests are applied at the early stages of the development. Test-cases may be created before the entire measurement system is built. For example, test-cases for the analog-to-digital converter can be created on the basis of technical specifications that define the system's operation. The development of the analog-to-digital converter can be done incrementally and it is completed when all the test-cases are performed successfully.

7.2. SeTT vs. DUnit test automation

Test-case "Bit-sync Locking" from Fig. 13 can be compared to automated test in Fig. 16. Fig. 16 presents the essential parts of automated test done in Delphi, using the DUnit framework. The complete test-case implementation in Delphi contains 111 lines of code (LOC), whilst the SeTT version contains definitions of 22 visual elements (11 building blocks and same number of connections), plus definition of variables, conditions, etc. One could argue, that looking from the perspective of test-case development effort, the advantages of SeTT are minor and that just the notation has been changed and visualised. However, these minor advantages in notation must be multiplied by the number of customers and the thousands of test-cases they have developed for their products. The most valuable asset is, that the domain experts can prepare test-cases in SeTT on their own without programming engineers interaction.

Table 2
Comparison between automated and manual tests in hours.

System	Manual	SeTT
Test 1	00:05	00:01
Test 2	01:28	00:40
Test 3	00:15	00:06
Test 4	04:45	02:47
Test 5	00:29	00:17
Test 6	01:40	00:56
Test 7	00:13	00:07
Test 8	08:39	01:43

7.3. Test automation development effort

From the development perspective, the entire testing tool SeTT comprises 23 kLOC, whilst the complete DEWESoft system contains 900 kLOC. In this way, the testing tool implementation represents only 2.5% of the whole measurement system. Looking closer at the DSML reveals that the syntactic and semantic parts of the construct "Assert" represent only 0.5 kLOC, while the Sequencer notation represents 24 kLOC. As seen on this example the development effort for SeTT was small and contrary to common belief that the effort for building such tools is probably huge. By building accompanying tools for concrete DSMLs, enough knowledge will be gained to enable us to automatically construct them. Current DSMEs support the automatic construction of editors and interpreters, whilst automatic generation of debuggers and test engines are rare.

8. Conclusion

DEWESoft customers are able to create a measurement procedure in a simple manner using a Sequencer, a domain-specific modelling language for data acquisitions. As in traditional software development, DSML can also return inaccurate results. In the measurement system, errors occur from various reasons (e.g., hardware, software). Finding them can be time consuming. To support end-users development and further improve the end-users productivity, the Sequencer has been upgraded with the testing features. In this paper a testing tool SeTT is presented within the DSML Sequencer. Tests are developed within a visual environment. Generality of usage was obtained with test definition at a metamodel level. Tests are used during the whole data acquisition process and able to test different system parts. The testing tool is already in use by many manufactures and suppliers who use a Sequencer as their measurement system.

Also, this paper presents recommendations that can help other researchers to reproduce DSML extension procedure (for example, testing infrastructure) for their DSMLs.

For the future work on SeTT and Sequencer we have several development directions. Regarding SeTT we plan to support automatic generation of tests similar to other model-based testing tools (Tedeso, TestStand) and assess the effectiveness of test automation (compared with native language tests previously defined in Delphi). Regarding Sequencer, most of development directions are connected with practical aspects reported by DEWESoft's end-users. For instance, we plan to develop a new modelling environment. We want to make modelling environment more friendly enabling several block interaction features (selection, movement, animations, zooming, connections, etc.). In this respect, a controlled experiment to compare end-user results with both versions of the modelling environment is also planned. We would like to see if the DSML notation has actual impact on the end-user results.

Supporting tools (e.g., debuggers, test automation) are often neglected by DSML developers. We believe these are essential for a DSML to be successful. Common tools from GPLs need to be presented more in DSME, too.

Acknowledgement

We would like to thank those DEWESoft co-workers who contributed to a discussion of DSML problems.

References

2009 Product of the Year Winners, 2010. NASA Tech Briefs. 34 (4), 10.

Allison, M., Morris, K.A., Costa, F.M., Clarke, P.J., 2014. Synthesizing interpreted domain-specific models to manage smart microgrids. J. Syst. Softw. 96, 172–193.

Atkinson, C., Kuhne, T., 2003. Model-driven development: a metamodeling foundation. IEEE Softw. 20 (5), 36–41.

Batory, D., 2006. Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Syst. J. 45 (3), 527–539.

Beck, K., 2003. Test-Driven Development: By Example. Addison-Wesley Professional.

Bringmann, E., Kramer, A., 2008. Model-based testing of automotive systems. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pp. 485–493.

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

Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G., 2011. Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. 8 (2), 225–253.

DEWESoft X2. Data acquisition software. Available at http://www.dewesoft.com/ last visited 17 September 2015.

Erdweg, S., Giarrusso, P.G., Rendel, T., 2012. Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12). ACM, p. 7.

Fewster, M., Graham, D., 1999. Software Test Automation: Effective Use of Test Execution Tools. ACM Press/Addison-Wesley Publishing Co.

Filho, R.S.S., Hasling, W.M., Budnik, C.J., McKenna, M., 2013. Experiences using Tedeso: an extensible and interoperable model-based testing platform. Autom. Softw. Eng. 20 (3), 299–337.

Fister Jr, I., Fister, I., Mernik, M., Brest, J., 2011. Design and implementation of domain-specific language easytime. Comput. Lang. Syst. Struct. 37 (4), 151–167.

Fowler, M., 2011. Domain-Specific Languages. Addison Wesley.

France, R., Rumpe, B., 2007. Model-driven development of complex software: a research roadmap. In: Proceedings of 2007 Future of Software Engineering. IEEE Computer Society, pp. 37–54.

Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J., 2008. DSLs: the good, the bad, and the ugly. In: Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19–13, 2007, Nashville, TN, USA. ACM, pp. 791–794.

Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J., 2007. Domain-Specific Modeling. Handbook of Dynamic System Modeling. CRC Press.

Hartman, A., Nagin, K., 2004. The agedis tools for model based testing. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 129–132.

Henriques, P.R., Varanda Pereira, M.J., Mernik, M., Lenič, M., Gray, J., Wu, H., 2005. Automatic generation of language-based tools using the LISA system. IEE Softw. 152 (2), 54–69.

Hermans, F., Pinzger, M., Van Deursen, A., 2009. Domain-specific languages in practice: a user study on the success factors. In: Model Driven Engineering Languages and Systems. In: Volume 5795 of Lecture Notes in Computer Science. Springer, pp. 423–437.

Hoyos, J.R., Garca-Molina, J., Bota, J.A., 2013. A domain-specific language for context modeling in context-aware systems. J. Syst. Softw. 86 (11), 2890–2905.

JUnit, Available at http://www.junit.org/ last visited 17 September 2015.

Karsai, G., 1995. A configurable visual programming environment: a tool for domain-specific programming. IEEE Comput. 28 (3), 36–44.

Kelly, S., Lyytinen, K., Rossi, M., 1996. 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), pp. 1–21.

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

Kieburtz, R.B., McKinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L., 1996. A software engineering experiment in software component generation. In: Proceedings of the 18th International Conference on Software Engineering, ICSE-18. IEEE Computer Society, pp. 542–553.

Knez, J., Tuma, M., Smith, G.M., 2002. A new approach to measurements – the PC instrument. Sound Vib. 36 (5), 16–20.

Kolomvatsos, K., Valkanas, G., Hadjiefthymiades, S., 2012. Debugging applications created by a domain specific language: the IPAC case. J. Syst. Softw. 85 (4), 932–943.

Kos, T., Kosar, T., Knez, J., Mernik, M., 2011a. From DCOM interfaces to domain-specific modeling language: a case study on the Sequencer. Comput. Sci. Inf. Syst. 8 (2), 361–378.

Kos, T., Kosar, T., Mernik, M., 2012. Development of data acquisition systems by using a domain-specific modeling language. Comput. Ind. 63 (3), 181–192.

Kos, T., Kosar, T., Mernik, M., Knez, J., 2011b. SeTT: testing-tool for measurement system DEWESoft. In: Proceedings of the 2nd International Conference on Design and Product Development (ICDPD '11). WSEAS Press, pp. 111–115.

Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M., 2008. A preliminary study on various implementation approaches of domain-specific language. Inf. Softw. Technol. 50 (5), 390–405.

Kosar, T., Mernik, M., Carver, J., 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empir. Softw. Eng. 17 (3), 276–304.

Kosar, T., Mernik, M., Gray, J.G., Kos, T., 2014. Debugging measurement systems using a domain-specific modeling language. Comput. Ind. 65 (4), 622–635.

Kulkarni, V., Reddy, S., 2003. Separation of concerns in model-driven development. IEEE Soft. 20 (5), 64–69.

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

Lindeman, R.T., Kats, L.C.L., Visser, E., 2011. Declaratively defining domain-specific language debuggers. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE 2011). ACM, pp. 127–136.

Luoma, J., Kelly, S., Tolvanen, J.-P., 2004. Defining domain-specific modeling languages: Collected experiences. In: Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM). Springer-Verlag, pp. 198–209.

Mernik, M., 2013. An object-oriented approach to language compositions for software language engineering. J. Syst. Softw. 86 (9), 2451–2464.

Mernik, M., Heering, J., Sloane, A., 2005. When and how to develop domain-specific languages. ACM Comput. Surv. 37 (4), 316–344.

Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.

Prähofer, H., Schatz, R., Wirth, C., Hurnaus, D., Mössenböck, H., 2013. Monaco – a domain-specific language solution for reactive process control programming with hierarchical components. Comput. Lang. Syst. Struct. 39 (3), 67–94.

Renggli, L., Gîrba, T., Nierstrasz, O., 2010. Embedding languages without breaking tools. In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10), pp. 380–404.

Saritas, H.B., Kardas, G., 2014. A model driven architecture for the development of smart card software. Comput. Lang. Syst. Struct. 40 (2), 53–72.

Schmidt, D.C., 2006. Model-driven engineering. Computer 39 (2), 25–31.

Seidewitz, E., 2003. What models mean. IEEE Softw. 20 (5), 26–32.

Selic, B., 2003. The pragmatics of model-driven development. IEEE Softw. 20 (5), 19–25.

da Silva, A.R., 2015. Model-driven engineering: a survey supported by a unified conceptual model. Comput. Lang. Syst. Struct. 43, 139–155.

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

Utting, M., Legeard, B., 2007. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann.

van Deursen, A., Klint, P., Visser, J., 2000. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not. 35 (6), 26–36.

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., 2013. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.

Wu, H., Gray, J., Mernik, M., 2009. 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, pp. 125–147.

Wu, H., Gray, J.G., Mernik, M., 2008. Grammar-driven generation of domain-specific language debuggers. Softw. Pract. Exp. 38 (10), 1073–1103.

Zhu, H., Hall, P.A., May, J.H., 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29 (4), 366–427.

Tomaž Kos received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2013. Currently, he works for DEWESoft company as a researcher. His main research interests include programming languages, domain-specific (modelling) languages, testing, data acquisition, and measurement systems.

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

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