Integrating Existing Proprietary System Models into a Model-driven Test Process for an Industrial Automation Scenario

Kai Beckmann
Distributed Systems Lab, RheinMain University of Applied Sciences, Unter den Eichen 5, D-65195 Wiesbaden, Germany

Keywords: MDSD, DSL, Metamodelling, Testing, MDT, Model-driven Testing.

Abstract: The introduction of modern model-driven software development methodologies into the industrial practise still proves to be a challenge. Especially small or medium-sized enterprises (SMEs) need an incremental and continuous modernisation process, which incorporates existing projects, is customised and cost-effective. Particularly, suitable solutions for model-based or -driven testing with test automation to increase the efficiency are in demand. This paper presents an approach for integrating existing proprietary system models of an SME partner for describing industrial automation processes into a model-driven test process, utilising a domain-specific language for the test specification. The test objectives focuses on the correct implementation of the communication and synchronisation of distributed state machines. The presented approach is integrated into a test framework, which is based on the Eclipse Modelling Framework (EMF) and the Eclipse Test and Performance Tools Platform Project (TPTP) framework. To separate the possibly changeable system and DSL-specific models from the implementation of the test framework, a stable and more generic test meta model was defined.

1 INTRODUCTION

While the model-driven aspect in software development is considered state-of-the-art and part of the curriculum of computer scientists/engineers, the comprehensive adoption in practise is still rare. This is especially true for small- or medium-sized enterprises (SME) of the embedded and industrial automation sector. Increasing software complexity necessitate a modernisation of the development processes and methodologies. Particularly the systematic testing is a pressing concern creating the demand for proper solutions.

Lack of time, missing knowledge, resistance from parties involved, the need to integrate existing projects and cost are reasons often heard why changes in software development are so challenging. The introduction of new processes, methodologies and tools requires training time for developers to become productive. Additionally, there is a social factor: developers have to relearn and accept new methodologies as beneficial. SMEs are often captured in the day-to-day business. There are seldom resources, such as man-power or money available to restart from scratch.

For SMEs, these problems result in necessary requirements for a successful modernisation of their software development processes. The modernisation process should be incremental and an introduction should be possible in the day-to-day business. There is a need to integrate and reuse existing projects, software development artefacts and models. Domain-specific adaptation to the existing development processes and projects can help to lower the effort for training and resistance offered of persons involved.

This paper presents an approach for integrating existing proprietary system models into a new model-driven test process facilitating a domain-specific language for test case definitions. The test process is going to be embedded in the development process of an existing project of a medium-sized company in the industrial automation sector. The existing system models are the result of an in-house development of the company for modelling state machines and code generation for various platforms. The test objects comprise the communication and synchronisation of distributed state machines, since these aspects are not part of the system model and implemented manually. This work is part of an ongoing R&D project with the company mentioned to develop a test framework for automated and model-driven tests.

In chapter 2 the necessary information about the properties of the system model and technologies em-

Beckmann K..
Integrating Existing Proprietary System Models into a Model-driven Test Process for an Industrial Automation Scenario.
DOI: 10.5220/0005240302550262
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 255-262
ISBN: 978-989-758-083-3

ployed is given. Chapter 3 starts with a brief outline of our solution and the overall architecture and presents the concept of the approach. The implementation is outlined in chapter 4, chapter 5 describes experience gained so far. Related work is discussed in section 6, and in section 7 a summary and an outlook upon future work are given.

2 BACKGROUND

2.1 Sequential Function Tables

Sequential Function Tables (SFT) are a proprietary table-based notation for state machines developed by Eckelmann AG (Wiesbaden, Germany) (Eckelmann AG, 2014). An example for a simple door control is given in Figure 1. States are noted in the columns and events in the rows. A transition is represented by an arrow from the start to the end state in the row of the triggering event. Transitions of the same event are merged in one arrow. In the example of Figure 1 the event *close*, representing the request to close the door, can trigger a state change from *Opening* and *Open* to the state *Closing*. Both arrows are merged pointing to the target state.

Furthermore, transitions can have a priority, which represents the order of evaluation of events during run-time. States can be ordered hierarchically into state groups. Transitions, and the entering or leaving of states or state groups, can cause actions, executed at run-time. The advantage of this notation is readability for its state machines with many states and transitions. As an example, state machines with more than 55 states and 45 events developed in practise are still manageable.

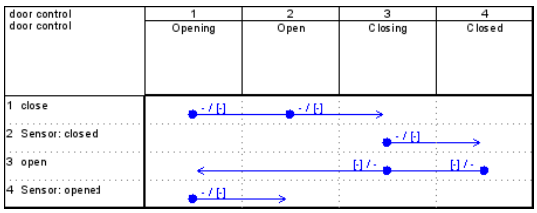

Figure 1: Example for SFT notation.

SFT system models are defined with the in-house tool StateCase, which generates code for various languages like C, C#, Java or even VHDL for FPGA targets. The code generation covers the structure and transitions of the state machines and code stubs for action and event processing which have to be implemented manually. The communication and synchronisation of distributed state machines is not part of the modelling and code generation process. Additionally, SFTs define state machine "classes" whose instances are deployed manually as well.

2.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) provides the modelling and code generation facilities for the model-based or -driven development of tools and projects with Eclipse (EMF, 2014). Furthermore, EMF is the core of other Eclipse-based tools and uses the meta-meta-model Ecore as common model basis. The EMF project provides an implementation of the OMG "Query View Transformation" (QVT) standard for specifying model to model (M2M) transformations for Ecore-based models (Eclipse QVT, 2014).

The Xtext framework supports the development of domain specific languages (DSL) for model-driven software-development processes (MDSD) within Eclipse and uses EMF as core (Xtext, 2014). By importing other Ecore-based meta models into the grammar, other models can be integrated and referenced.

The Eclipse Test & Performance Tools Platform Project (TPTP) (TPTP, 2014) was initiated by IBM in 2002 as the "Hyades" project and provides Eclipse plugins to develop and execute unit and performance tests for applications with Eclipse. The main use case is the test of Java applications, but the framework provides extension points to adapt it to other environments and domains. Currently, the project is not actively being developed and is preserved in the last stable version.

2.3 Test Modelling Standards

TestIF is the Test Information Interchange Format, a currently finalised standard of the OMG for the "exchange of test information among tools, applications, and systems that utilize it" (OMG, 2014). The standard defines a platform-independent meta model. To exchange test information, the purpose and the structure of tests can be defined, as well as the test data and expected responses of the system under test (SUT). Furthermore, the results and artefacts of the execution of tests can be specified as well.

The UML testing profile (UTP) is a UML profile to extend UML with semantic elements for test modelling (Baker et al., 2008). UTP provides the semantics to model test architecture, behaviour, data and results using UML. Other UML profiles can be incorporated as well, so it is possible to associate requirements from SysML with UTP test objectives. Besides the profile, the UTP standard defines a MOF-based meta model as well.

3 CONCEPT

3.1 Use Case & Requirements

In this use case the automated and model driven testing is introduced in a project at the end of its active development cycle. The SUT consists of several distributed state machines running on heterogeneous platforms, like control systems, micro-controllers and FPGAs. The overall application must satisfy real-time and safety constraints. SFTs are used to specify the structure of the state machines.

In the original state of the project, a commercial tool is used for the acquisition and management of application requirements and related test cases. A variety of mock-ups and simulators have been developed for manually executed integration and system tests. The manual tests consist of sequences of consecutive test steps. Test steps can either drive the test by operating the simulators and mock-up components, or instruct the tester to validate the correct SUT behaviour. The interaction of all test components is complex and the manual test execution is very time consuming. Therefore, regular and complete regression tests are not feasible in practise.

Through discussions with our industrial partner the following main requirements were identified:

- Need for a test DSL, adjusted to the existing manual test specification and structure
- Reuse of the existing system models for test specification
- Model-driven process with the usage of established software standards
- Integration in a test framework for test specification, management, execution and result evaluation/reporting
- Reusability of the approach and generic implementation for other projects and industrial users
- Usage of Open-Source tools

3.2 Approach

To meet the requirements identified with our industrial partner, an approach based on a model-driven test process within a test framework was developed, whose overall architecture is depicted in Figure 2. The existing SFT-based system models are imported and reused in the test specification, for which a DSL for distributed state machines was developed.

One requirement for this approach is that it should be reusable for other (at least related) use cases and projects. Therefore, the meta models of the DSL and proprietary system model are not directly employed in the test process, since they are too platform and use case specific. Instead, an overall meta model, named TEFAModels, was developed, which merges system and test model aspects. Model transformations convert the DSL models and proprietary system models to a TEFAModels representation, thereby decouple them from the test framework. This also allows to replace the currently textual DSL with other possible notations, like a graphical version. Changes of the system model are limited to the transformation as well, if an adaptation of TEFAModels is not required.

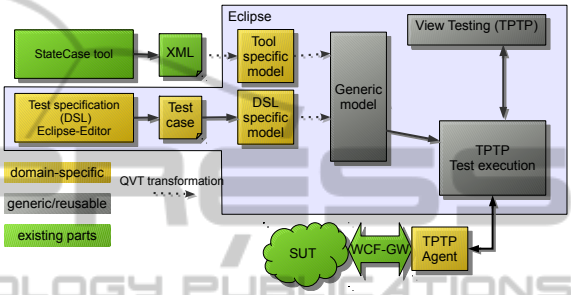

Figure 2: Overall architecture of the test framework.

In TEFAModels, tool and platform specific model elements are separated from platform independent aspects, which simplifies extensions and adaptations. The structure and the scope is oriented on other established software and test standards. However, TEFAModels is not intended to be a completely generic test meta model for any possible use case or software application. Reasonable reusability is limited to comparable scenarios with distributed state machines and a sequential test step execution. Nevertheless, these constrains still allow a wide applicability in the industrial automation sector.

The Eclipse Test and Performance Tools Platform (TPTP) serves as a foundation for test management and execution and was adapted to the developed meta model. The Eclipse Modeling Framework (EMF) provides the facilities for the model-driven process. The test interface to the SUT is realised by a gateway developed by the industrial partner (WCF-GW in Figure 2). Active state machine instances can be queried and events of state transitions subscribed. This interface is operated by a TPTP agent, which relays information and commands between the SUT and the test framework.

3.3 Integration of the Proprietary System Model

In the presented approach the proprietary system model is decoupled from the meta model used in

the test execution. Therefore, the proprietary system model has to be imported or parsed into a processable representation and then transformed to a more generic model as part of the model-driven test process. In such a system meta model, all information relevant to the test of the proprietary system model has to be representable.

The reason to use an approach with an intermediate model transformation step can be illustrated by the use case presented in this paper: in this use case, the state machines of the system are modeled with the tool StateCase and stored in a proprietary XML format. An XML schema can be mapped to a corresponding meta model. Figure 3 shows a fragment of the resulting meta model structure for the StateCase format, using the EMF facilities for this task. The internal StateCase-specific structure, looped through the XML format, becomes evident; IDs are used to reference elements like states, events or transitions. Inheritance is not used.

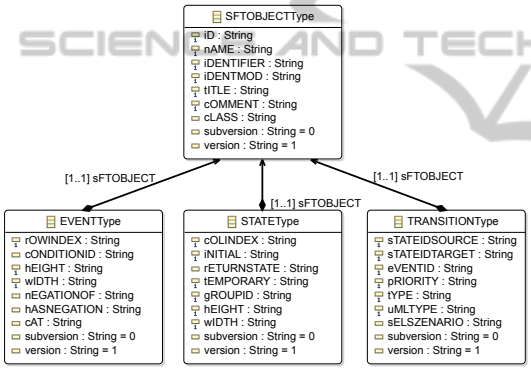

Figure 3: Generated meta model for SFT.

The usage of a model structure like this would complicate the further model-driven test process. Only the effort to resolve references during runtime can justify a transformation into a more applicable model. Here, the meta model TEFAModels was developed to provide a more generic model for distributed state machines. The structure, displayed in Figure 4, is derived from a simplified version of the UML model for state machines (OMG, 2011), since it is well established and meets the requirements.

From this platform-independent meta model of state machines, SFT-specific elements are derived, enriched with the information of the SFT-models not generically representable and necessary for the test specification and execution. Thereby, a PIM and PSM layer are established.

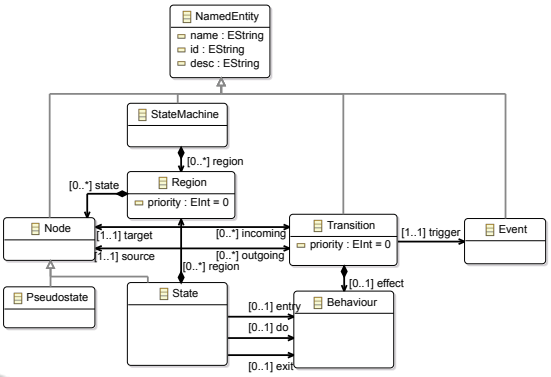

Figure 4: Generic meta model for state machines, derived from UML.

3.4 Domain-specific Test Modelling Language

A DSL should be specific for the given use case. In the approach presented in this paper the existing system model is reused for the test specification with the DSL, for example to define a predicate over the order of occurring transitions of state machines specified in the system model.

In this use case a DSL with a textual C-like syntax was developed, because the testers of the cooperation partner are developers familiar with a representation like this. In other use cases or with different domain experts this might not be favored. Though, with the decoupling of the DSL meta model from the more generic TEFAModels representation, it is possible to provide different DSLs, specific for the given needs, unified in the transformation process.

In the following part of this section the characteristics of the developed test DSL for our project partner are presented. The test objective is the validation of the manual implementation of the generated state machine stubs, in which the communication and synchronisation between distributed state machines is realised. The chosen structure reflects the requirements of the project partner, the capabilities of the test system and especially the sequential execution of the tests.

Tests are organised in test suites which contain a set of consecutive test cases. A test case itself consists of a sequence of test steps and should have pre- and post-conditions. These specify the state the SUT has to have before and after a test case is executed. In addition, conditions can be defined for test steps as well. If a pre-condition is not met, the test case is aborted and the next coequal step in the sequence is executed. It is possible to define alternative sequences to remedy previous abortions.

A single test step can either interact with the SUT to drive or control the test execution or validate the SUT behaviour. A test is driven by forcing events to trigger transitions of state machines. With this mechanism, a SUT can be pushed into any reachable state. Occurring transitions are monitored and used for the behaviour validation of the SUT.

The expected SUT behaviour is modeled using predicates over permitted paths of state transitions. These paths can span multiple state machine instances including distributed configurations to reflect their interaction with each other, which we call "global paths".

```
Assert {
  PathAssert {
    sm1:state_1 -> sm2:state_a -> *
      -> (sm1:state_2 or sm1:state_3)
  } timeout 300
  PathAssert {
    not sm1:state_4
  }
}
```

Listing 1: Example for behaviour specification.

A small example of a behaviour definition with two state machine instances `sm1` and `sm2` is given in Listing 1. A global path is defined from the *state_1* of the `sm1` to the *state_a* of `sm2`. As next step in the path, a wildcard specifies that any transition of any state machine instance is acceptable if as next step the path ends in the *state_2* or the state *state_3* of `sm1`. This behaviour has to be observed within 300 ms, otherwise the test fails. In parallel, the second PathAssert shall be evaluated during test execution, in which the occurrence of a transition to the state *state_4* of `sm1` would lead to a failure of the test.

The modeling elements of the example can be joined as needed; the timeout can be specified optionally. For the given use case and the current development state, these expressions are sufficient. An extension regarding more detailed real-time constraints is under construction.

As a constraint of the test execution environment the given test harness interface to the SUT does not allow access to internal variables of the SUT. Therefore, the expressiveness of the test specification is limited to states and transitions of the state machines. Since the system model does not contain information about the state machine instances and their deployment, this information has to be specified as a prefix of the test specification.

3.5 Generic Test Meta Model

The purpose of the developed meta model TEFAModels is to provide a stable model foundation for the test framework. Besides the state machine based system model from section 3.3 and the test information defined by the DSL, it contains necessary aspects of test management, test structure, execution artefacts and test results. This meta model was not defined from scratch, but uses concepts of existing and related standard models. Thereby it can profit from the experience of other projects and might ease a future transformation into these established representations. The main sources are TestIF, UTP and the internal meta model of TPTP.

The basic structure of the tests in the meta model in the form of sequences of test cases, test steps and test sets is adopted from TestIF. Regarding test results, the UTP verdict definition is used, extended by the more detailed logging and justification elements of the TPTP meta model.

The reason not to use the meta model of TPTP as primary foundation is the tight coupling of the meta model to the testing of object-oriented application software. Furthermore, the information needed for the validation of a test case is not part of the meta model, but encapsulated by domain- and platform-specific implementations. An adaptation to the given use case was not considered reasonable. As a drawback of this decision, parts of the TPTP test execution implementation had to be reimplemented.

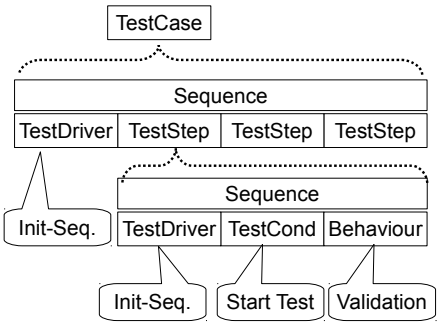

Figure 5: Example for a test case structure.

The test structure of the TEFAModel is similar to TestIF, with the exception that a test suite represents a single encapsulated test model instance. A *TestSuite* contains the test cases with expected behaviour, the deployment and configuration on the SUT and the results and traces of executed tests. There are two kinds of test objects: sequence objects and atomic entities. Sequence objects like *TestCase*, *TestStep* and *TestDriver* can contain a list of other test objects, whereas atomic entities as *TestDriverStep*, *TestCon-*

dition and *BehaviourDefinition* can only be enclosed by a sequence object. The test objects can be combined in any order as needed, but a basic structure, as exemplary displayed in Figure 5, is needed for the test execution. A *TestCase* should contain a sequence of *TestDriver* and *TestStep* objects. These objects can contain other sequences or *TestDriverStep* and *BehaviourDefinition* objects. The latter models atomic interaction with the SUT or assertions for the behaviour validation. As described in section 3.4, the execution of the test traverses the sequence and evaluates the elements. For this example, the SUT is initialised with one driver sequence and another one triggers the start of a validation step.

Given the restrictions of the use case towards the test interface to the SUT, the SFT-specific derivations of *TestDriverStep* are limited to force events of state machines. To control the test execution, *TestCondition* derivations can be inserted in the sequence halting the process until certain conditions are met, e.g., a state machine has entered a defined state.

The expected SUT behaviour is not specified in detail in TEFAModels, since the validation during the test execution is performed by generated code. Therefore, the SFT-specific derivation of the *BehaviourDefinition* contains the path to the generated Java file only, which has to be instantiated during run-time.

The execution of a test is represented by a *TestRun* object. Each test run references the test object used, results and time of execution. The monitored behaviour of a SUT is represented in a trace of state transition events. These events reference the corresponding state machine elements of the system model and contain the required information about their occurrence, such as time-stamps, sequence numbers etc. A trace event is assigned to the active test runs. A test result contains the common verdict information if a test has passed, failed or is inconclusive. Additionally, a *VerdictReason* similar to the TPTP meta model can specify the reason in more detail.

4 REALISATION

For the prototypical realisation of the test framework, several plugins for Eclipse and TPTP have been developed to integrate StateCase, the new test modelling environment and the test execution. The integration of the StateCase XML files uses the EMF facilities, and all meta models are based on Ecore. The DSL is specified and the tool support realised using the Xtext framework. All model transformations are described and conducted by the QVT Operational implementation of Eclipse. To be able to use the Kepler (4.3) version of Eclipse and meta modeling tooling support, the necessary TPTP plugins were ported from the Helios version of Eclipse, which is presented in a paper for a sibling project (Thoss et al., 2014).

Based on the DSL grammar, the Xtext framework generates a corresponding meta model, a parser and a comfortable editor for Eclipse. For integrating the existing system models, the grammar has to import the system meta model discussed in section 3.3, and it has to reference the state machine elements. An Xtext resource service provider has been implemented to provide access to the system models for the generated Eclipse editor. As a result, the test models directly reference elements of the imported system models. Furthermore, the editor provides appropriate suggestions and autocompletion during test specification.

The test structure and deployment information of the DSL are transformed to a TEFAModel instance which can be processed by the test framework for test execution. This transformation, realised with QVT, uses the DSL meta model as source and has to be adapted if the DSL changes in the future.

The expected SUT behaviour is not modeled in TEFAModels yet and therefore not transformed. Instead, the Eclipse Xtend language is used to generate the Java code for non-deterministic finite automata (NFA), representing the specified assertions or predicts. The NFA is part of the test oracle and dynamically instantiated during the test execution. The NFA consist of all possible valid state graphs the predicts or assertions allow for a validation step. During test execution, the test framework monitors the SUT state transitions and relays these events to the test oracle, which drives the contained NFA. If an NFA enters a valid final state, the test passes. On the other hand, a test fails if reaching a final state becomes impossible or a timeout occurs.

5 EVALUATION AND LESSONS LEARNED

As part of an iterative development process, several versions of the test framework have been evaluated by domain experts of our industrial partner. The feedback led to changes of the requirements and necessary feature sets. Focus in the first iterations was on usability and expressiveness of the test specification and the test process. The version considered in this chapter comprises the whole model-driven test specification process and the connection to the SUT's test interface, integrated in Eclipse and TPTP. The detailed reporting of the test results is planned until end of 2014.

The (re-) modelling of the existing manual test us-

ing the DSL and the facilities of the test framework turned out to be difficult at the beginning. To conduct a manual test, the tester interacts with several test components and external applications. The integration or control of these components was not considered at first, since the heterogeneous interfaces required significant effort to automate them. As a consequence, options to call shell scripts and to interact with the tester during test execution were added as requirements. Furthermore, the expressiveness of the test DSL and the automated test execution allows a much finer-grained control over tests, which the testers were unaccustomed to. As a result, the developed test cases were relatively simple initially, but with increasing familiarity the testers approved of the new potential.

The introduction of automated test execution showed the typical improvements of reduced execution time and deterministic test repetition in comparison to the previous manual test execution.

The integration of the existing system model and its reuse for test specification is positively appreciated by the domain experts. It supports the development of tests and reduces errors and mistakes. Especially the tooling integration with autocompletion and suggestions based on the imported system model is reckoned as beneficial.

TPTP had to be adapted and customised more than expected, mostly because of the TPTP meta model. As motivated in section 3.5, an adaptation of the meta model was not consisted reasonable. Therefore, the integration of our TEFAModels required significant implementation efforts. The parts of TPTP which references the TPTP meta model in the source code had to be adapted or reimplemented. Most effort went into test execution, since our approach differs not only by the meta model, but also in the basic test structure. On the other hand, this also permits a much deeper control of the online test execution and validation, specifically for the test of distributed state machines, than the given TPTP implementation. In general, the chosen framework architecture and the separation of meta models in the model-driven approach proved to be beneficial. The test framework can be adjusted to changes and additions easily. Four new StateCase versions were introduced during the development. Adaptations were limited to the regeneration of the XML-to-Ecore mapping and to modifying small parts of the QVT transformation. Even if new attributes or information are introduced, the separation of platform-independent from platform-specific models in TEFAModels limits the required changes.

6 RELATED WORK

Domain-specific languages, having been an established methodology for a long time, became quite popular recently. The main advantage of DSLs are the “improving [of] productivity for developers and improving [of] communication with domain experts” (Fowler, 2010). As part of the model-driven software development, DSLs simplify the specification of formal models and the usage for domain experts (Stahl et al., 2006). Therefore, there are several successful applications of DSLs for test specification purposes in literature and practise. Several model-based testing tools are using some flavor of domain-specific language or model for the test model specification (Shafique and Labiche, 2010; Mussa et al., 2009).

The selection of a proper testing language and test approach depends on the use case, the test organisation and the SUT. In (Hartman et al., 2007) this problem area is surveyed, and the authors provide guidelines to support the decision depending on the specific environment. It is pointed out that DSLs have many advantages, but the implementation causes significant effort and needs specialised development knowledge, especially for in-house solutions. It is stated that the usage of open source projects lowers the effort and reduces the dependency on third-party vendors.

In (Wahler et al., 2012) an approach similar to ours is presented. The use case is the development of an automated testing system for “a software library for an embedded real-time controller used in automating processes”. Existing manual tests which take over a person month to execute, were remodeled with a DSL and automatically executed. In contrast to our use case, the test system can access internal variables, events and alarms of the SUT by an OPC-server, whereas our test interface is limited to the monitoring of transitions of state machines.

Furthermore, the existing variables and possible events of the SUT are queried from the OPC-server and made available in the test specification editor. In our approach the existing proprietary system model is integrated and therefore, the test specification phase is independent from the access to the SUT. Nevertheless, the utilisation of an OPC-server and the possibility to access process variables allows more comprehensive predicates over the SUT behaviour. Since the usage of OPC is common in the automation context, the authors’ realisation can certainly be applied to other use cases. Another similarity is the usage of Eclipse and the Xtext framework for the realisation. However, the authors implemented the test framework from scratch and the test execution is realised with Scala using the AST representation of the test model.

In our approach, we reused and adapted the TPTP framework and we use stable Ecore-based models for the test specification and execution process. Changes in the DSL have to be respected in a model transformation and do not necessitate adaptation in the test execution implementation.

7 SUMMARY, FUTURE WORK

The paper presented an approach for integrating existing proprietary system models into a new model-driven testing process. For an industrial automation scenario, a domain-specific language was developed to specify test cases for a SUT consisting of distributed state machines. As part of the model-driven test process, a meta model consisting of a more generic representation of the system model and necessary testing aspects was defined. The existing system model and the DSL are integrated in the testing process by model transformations which decouple the domains, keeping changes and adaptations locally. The realisation is based on Eclipse and the Eclipse modelling framework. As a foundation for the test framework, the TPTP project was adapted and customised to the use case. The iterative development with continuous evaluation of intermediate results by domain experts ensured the practical usability and acceptance by the testers.

The version of the test framework presented in this paper lacks the complete reporting and evaluation capabilities of test results, which is ongoing work. Furthermore, the DSL and test evaluation is currently extended with real-time capabilities to enhance the usability of the test framework further.

So far, the defined test meta model has generic parts, but is limited to a use case based on state machines. Thus, it is a specialised solution. As a next step, the TEFAModels will be used as an intermediate model for a transformation into a UML/UTP based representation. Besides the possibility to exchange and reuse these models, adapting the test framework to execute and manage tests represented using these standards will result in broader applicability.

ACKNOWLEDGEMENTS

The master students Matthias Jurisch and Michael Poetz, my colleague Marcus Thoss and Horst Wiche and Matthias Englert from the Eckelmann AG contributed to the presented work. Furthermore, I want to thank my Ph.D. supervisors Prof. Dr. Kroeger from Rhein-Main University of Applied Sciences and Prof. Dr. Brinkschulte from Goethe University Frankfurt am Main.

This project (HA project no. 317/12-07) is funded in the framework of Hessen ModellProjekte, financed with funds of LOEWE – Landes-Offensive zur Entwicklung Wissenschaftlich-Oekonomischer Exzellenz, Foerderlinie 3: KMU-Verbundvorhaben (State Campaign for the Development of Scientific and Economic Excellence).

REFERENCES

Baker, P., Dai, Z. R., Grabowski, J., Haugen, O. y., Schieferdecker, I., and Williams, C. (2008). *Model-Driven Testing: Using the UML Testing Profile*. Springer.

Eckelmann AG (2014). http://www.eckelmann.de/en.

Eclipse QVT (2014). Eclipse Model to Model Transformation Project. www.eclipse.org/mmt/qvto/.

EMF (2014). Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/.

Fowler, M. (2010). *Domain-Specific Languages*, volume 5658. Pearson Education.

Hartman, A., Katara, M., and Olvovsky, S. (2007). Choosing a Test Modeling Language: a Survey. In *Hardware and Software, Verification and Testing*, volume 4383 of *LNCS*, pages 204–218. Springer.

Mussa, M., Ouchani, S., Sammane, W. A., and Hamou-Lhadj, A. (2009). A Survey of Model-Driven Testing Techniques. In *9th International Conference on Quality Software, QSIC '09.*, pages 167–172, Jeju, Korea (South). IEEE.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure.

OMG (2014). Test Information Interchange Format 1.0 - Beta 2. http://www.omg.org/spec/TestIF/1.0/Beta2/.

Shafique, M. and Labiche, Y. (2010). A Systematic Review of Model Based Testing Tool Support. Technical report SCE-10-04, Carleton University.

Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Thoss, M., Beckmann, K., Kroeger, R., Muenchhof, M., and Mellert, C. (2014). A Framework-based Approach for Automated Testing of CNC Firmware. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, JAMAICA 2014, pages 7–12, New York, NY, USA. ACM.

TPTP (2014). Eclipse Test & Performance Tools Platform Project. http://eclipse.org/tptp/index.php.

Wahler, M., Ferranti, E., and Steiger, R. (2012). CAST: Automating Software Tests for Embedded Systems. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 457 – 466.

Xtext (2014). Eclipse Xtext Project. http://www.eclipse.org/Xtext/.