Applying Test-Driven Development for Creating and Refining Domain-Specific Modeling Languages and Generators: Demo

Juha-Pekka Tolvanen

MetaCase
Ylistönmäentie 31, FI-40500 Jyväskylä, Finland
www.metacase.com
jpt@metacase.com

Abstract

While creation of modeling languages and related code generators has become easier, their testing is perhaps nowadays the most time consuming part. This is particularly relevant for Domain-Specific Modeling (DSM) as it is common that the language and generators evolve frequently along with the domain it addresses. We demonstrate one solution that we have found practical when applying Test Driven Development for creating and maintaining modeling languages and generators along with tool support.

Categories and Subject Descriptors D.2.2 [Software Engineering] Design Tools and Techniques - state diagrams D.2.6 [Software Engineering] Programming Environments - programmer workbench, graphical environments D.3.2 [Programming Languages] Language Classifications - Specialized application languages

General Terms Design, Languages, Verification.

Keywords test-driven development; domain-specific modeling; metamodeling; generators, modeling languages

1. Introduction

Modeling languages that raise the level of abstraction, yet generate code and other artifacts needed, have proven to improve significantly the development productivity and product quality [1]. Recent empirical research shows that companies who successfully apply modeling and generators tend to create their own domain-specific languages [2]. One reason for this trend is Language Workbenches. These tools enable creating modeling languages easier and faster than ever: Data from several cases [3] shows that we can create a language for a given task in a few days, use it as long as the project runs and then roll-out a new language for the next project. Today testing a DSM solution has perhaps become the biggest effort – in particular when the domain evolves and the language is updated more frequently than typical for general-purpose languages. On the other hand, being able to fully control the language and knowing about its use gives also the additional opportunity to automatically migrate work done to the new language version. We have been involved in creating several DSM solutions and have found Test-Driven Development (TDD) particularly suitable on both language and generator side. We describe the approach and demonstrate its use with an example.

2. Example domain

We illustrate the approach with an example of Internet of Things (IoT) device. Rather than writing the application code manually we created a DSM solution with two modeling languages and a JSON generator[1]. A fully functional and completely executable code can be generated from the models and run in an IoT device[2].

The core language is based on a state machine that is extended with services and rules of the device. Sensors (movement, humidity, GPS, etc.) provide various events along with some internal household features like checking battery level and charging status. Action part of the state machine covers the various communications the device can send to the outside world (like messages to cloud, SMS). Figure 1 shows an example application for monitoring the boat in a harbor using the modeling language. This application checks the humidity level every hour and sends warning to the phone if humidity is above 55%rh. The application also checks the temperature and warns if temperature is approaching the freezing point. Most importantly the application checks the boat location and warns if it leaves the harbor area and sends then tracking information to the cloud. The second language enables developing several integrated applications (Figure 4) with the first language.

3. Test-Driven Development Applied

Before implementing the language we inspected the domain and identified core language concepts and tried to apply them to create some typical IoT applications. Next, and following TDD, we define a library of very small applications that aim to use all language concepts – at least in one case. These minimal applications serve as test cases for the DSM language. For the same minimal applications, we wrote the expected code implementing them. These form the test cases for the generators. We exclude thus testing of the external framework and application execution.

---

[1] http://www.metacase.com/blogs/jpt/blogView?entry=3620043178

[2] https://www.thingsee.com/

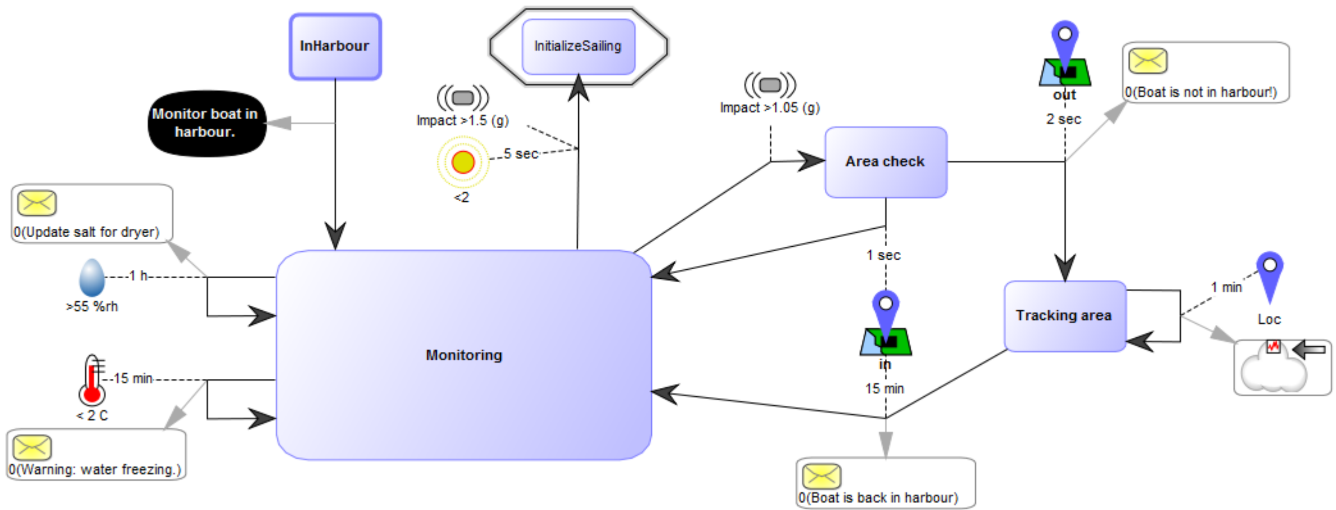

Figure 1. Harbor monitoring application specified with the DSM

3.1 Test models for the language

When implementing the language, its concepts, constraints and notation, we created the same test case models with the newly created language. Since our example language is based on a state machine we started with a simple case: A transition with two states and defined this metamodel within MetaEdit+ tool [4]. After that the related test model was immediately specified in the same tool. Figure 2A illustrates this minimal test model. Language definition then continued by adding all the other language concepts from the library to the metamodel and creating the respective test models (e.g. Figure 2B-2D).

We also created alternative test models for the core state machine, like a transition targeting the same state, multiple transitions from a state, multiple transitions to a state, a transition with and without events etc. Having the test model for the core part, adding the remaining test models covering other sensors (12 in a total) and actions (7 in a total) was straightforward.

Our test models did not cover all possible cases as the number of alternatives would be too big. For example, when reading data from a sensor it could be decided if the value is saved to the device, how often the value is polled from the sensor, and in each case large number of alternative actions was possible (31 different). While work has been done to automate language testing [5, 6], we created manually the test models covering the typical cases (e.g. sensor reading intervals: sec, min, hour etc.) as well as cases if sensor value is saved to a device. This way all model instances got different values, but not all of their combinations. This resulted into 37 different test models.

While creating the test models we also tested the language definition made by trying to create models that address the language rules. In other words, tried to create models that have more than one start state, a transition that tries to trigger a sensor, an action that activates a sensor etc. which should not be possible based on the metamodel.

3.2 Expected code to be generated

For developing and testing the generator, a special ‘Expected code’ block was added as a modeling element for each of the 37 test models including the expected output from each model. The code content was taken from the original test cases, or if not already available, defined before making the respective generator. After this, a generator was implemented for the added metamodel concepts extending the generator functionality gradually. Figure 3 illustrates this approach showing the expected code next to the test model: The test defines the case of reading speed data in 2 sec interval and triggering transition if speed is more than 10 km/h but less than 100 km/h.

To support testing of pretty printing we added to the ‘Expected code’ block the actual generator output produced by the generator. This way both expected code and generated code could be seen at the same time, and overlaying them with different font colors the human eye could spot the differences quickly.

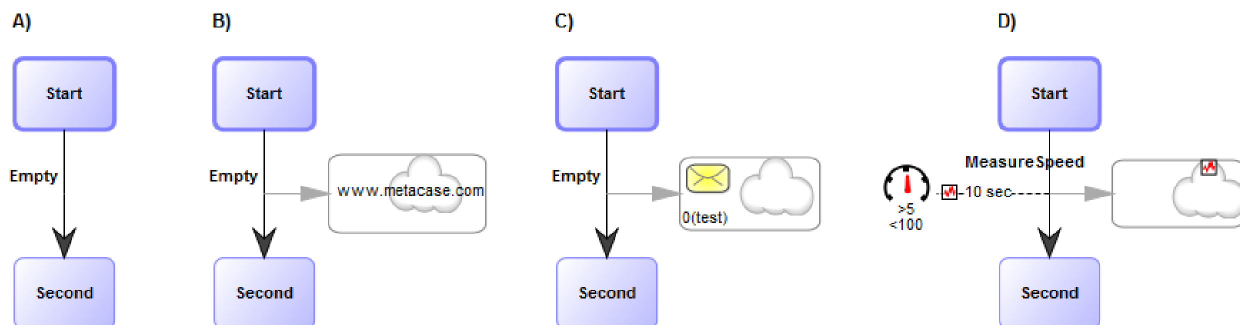

Figure 2. Test models covering four test cases

Figure 3. Test model for measuring speed

3.3 Running and automating tests

When the number of test models increases, automating their execution becomes useful. A simple way is to generate code from a model along with the expected code to files for a comparison. Rather than doing it for each test case we can run it for all test models at the same time too. In our case of IoT language we applied the language for developing several integrated applications for this purpose. Figure 4 shows the language used to create a library of all test models - in our case for 37 test models.

Each time a generator is added or modified, we can run all test cases at once supporting regression testing. This is perhaps the biggest single value from the approach as it provides evidence that the generator changes do not break generators targeting other parts and other test cases. In our example, we restrict the testing to the generated code only as the target platform is taken as given. If we would need to test also a platform, TDD approach could be extended towards executing generated code along with the existing code or framework.

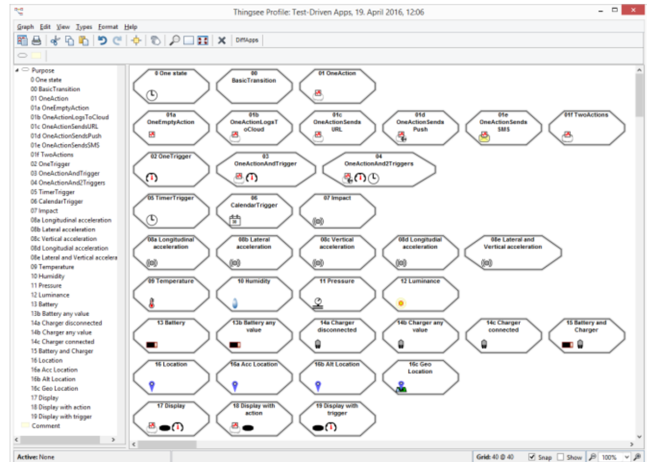

Figure 4. All test models

4. Discussion

Use of TDD makes the process of creating a DSM solution controlled and systematic. During DSM implementation it becomes easier to estimate what is done, what needs to be done, and most importantly ensure that what is defined can be validated. This is particularly important as domain always evolves. We see that TDD approach could also help in dividing DSM creation work as collaborative language workbenches (e.g. MetaEdit+) enables working parallel: While language engineer defines test models, the generator developer can extend them with test cases for the generator.

We have not applied TDD for all parts of DSM solution but see that naming rules (e.g. via RegExp as in [4]) and translators during code generation could be addressed too. We see that testing could be easily extended with static code analysis and executing the code with the framework. In the industrial cases where we have applied TDD approach, as demonstrated here, the effort in the beginning is not considered significant and the iterative approach for development and automating tests pays off quickly.

Acknowledgements

We thank Jennek Geels, Teemu Kanstren and Olli-Pekka Puolitaival on their proposals and feedback to this work.

References

[1] Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D., "What Kinds of Nails Need a Domain-Specific Hammer?", IEEE Software, July/Aug, (2009)

[2] Whittle, J.; Hutchinson, J.; Rouncefield, M., "The State of Practice in Model-Driven Engineering", IEEE Software, vol.31, no.3, (2014).

[3] Tolvanen, J-P., Kelly, S., "Model-Driven Development Challenges and Solutions: Experiences with Domain-Specific Modelling in Industry", Proceedings of the 4th Modelsward, SCITEPRESS, (2016)

[4] MetaCase, "MetaEdit+ User's Guide", (2016)

[5] Baudry, B., et al "Model Transformation Testing Challenges". Procs of the IMDDMDT workshop, (2006)

[6] Kanstrén, T., Chechik, M., Tolvanen, J-P., "A Process for Model Transformation Testing", In: Systems Testing and Validation, (2015)