Testing an ASIC: an mbeddr Case Study

Daniel Stieger[1] and Michael Gau[2]

[1] die modellwerkstatt/itemis
[2] Bachmann electronic GmbH

Abstract. Last Changed: Feb 22, 2013

1 Contacts for more Info

The asic testing system is developed by Daniel Stieger and Michael Gau, reachable via `daniel.stieger@modellwerkstatt.org` and `m.gau@bachmann.info`.

2 What is the problem/domain your system addresses?

An ASIC is an electronic chip specifically designed for a particular purpose. Compared to common integrated circuits (ICs), an ASIC (short for Application-Specific Integrated Circuit) is a semiconductor device customized for a particular use, rather than intended for general-purpose use. Typically, an ASIC has only a single customer, which is exactly the one who led design the chip to put it into his products.

One of our customers ordered a quite complex ASIC design, providing various I/O functionalities such as analog/digital conversion, filters and counters. The ASIC is used in an I/O module for control systems where application engineers can configure any functionality needed for their applications.

One of the most important features the I/O module provides is an extensive self test of the ASIC. Therefore, the application engineer can put a test connector on the I/O module, wiring the output ports to its own input ports. With this setup, the control system can perform a self test of the I/O module which consists of approximately 60 cycles of the following procedure:

1. reset chip
2. configure functionality
3. write value to output
4. read value form input and compare it to expected value

The developers of the ASIC came up with an Excel spread sheet that described the tests. Each row described one cycle with all configuration details; write- and expected read values were stored in the columns. What was needed next was a translation of the Excel file into plain C-code, which is executable by the control system.

3 How does the system integrate/use mbeddr's existing capabilities?

In a first step, an MPS-language for the test descriptions was created. A single test (or cycle) consists of a description, configuration settings, values to be written to the specified output ports and expected values at the respective input ports. Additionally, a description of the problem that was likely to cause a particular failure was formulated.

In a second step, the Excel file was saved in CSV (comma separated values) in order to easily read it with a few lines of Java File-IO. This Java code was plugged into an MPS refactoring: the refactoring imports the data from the CSV file and built a model in the MPS language that describes the tests.

In a third step, we used mbeddr's code-generator to translate the test descriptions (cycles) into plain C-code. mbeddr allows importing existing header files, which describe the environment of the controller system. While writing the code-generator, code completion, syntax checking and everything else known from sophisticated IDEs is available, simplifying the task of writing the generators.

4 Why was mbeddr/MPS chosen as the basis?

Our problem relates strongly to code generation. While there are various language workbenches that support Java, to our knowledge only mbeddr/MPS has strong support for the C language (of course, MPS supports Java as well). Without support for C, we would need to write a code-generator in the manner of concatenating strings. Indeed, it was discussed to write a Visual Basic Macro to translate the test descriptions to C code. Without any doubt, that would have result in a very error prone procedure, especially when considering that number of configuration values and read/write values differed slightly (resulting in quite different C code per test).

mbeddr/MPS allows for the development of the code-generator itself with full C IDE support, that is not only syntax highlighting but also syntax checking. This makes the code generator very maintainable: it is easily readable has has a clear and understandable structure. From the beginning, the whole code was directly developed in the code generator itself. Necessary definitions were imported from header files and referenced in the generator. For us, this is quite the opposite of writing code-generators where the development environment understands strings only.

Our mbeddr-based code generator automatically generates the necessary C and header files for our self-test. No further adjustments were need to the generated code. The code was directly referenced by the build system and could be re-generated after changes in the test description at any time. mbeddr/MPS allowed us to implement full round-trip engineering.

mbeddr/MPS supports refactorings. A refactoring is a piece of Java code, which can access and manipulate a model directly. This makes it quite easy to change existing code, or – as we did – build a model from scratch by importing information from another source.

Most important, mbeddr/MPS dramatically accelerated the development by reducing complexity and installing a common language. As already described, in a first step a suitable test-language was discussed and implemented, which led in turn to a common understanding of the used concepts. Taking advantage of this clearly defined concepts enhanced communication between team members.

5 What is your conclusion?

Based on our experience, mbeddr and MPS is a very appropriate tooling for Model Driven Software Development (MDSD) when working in C and Java environments. Java and C are extensively supported by the IDE, not only when writing solutions or models, but also when developing code generators. Using mbeddr C in our code generator prevented us from having any syntax errors in generated code – starting with the first iteration.

The idea of custom refactorings was very appealing to us. With mbeddr and MPS one can manipulate a model very easily and – of course – with full IDE support. Refactorings are basically written in Java, any Java package can be accessed including file-io and access to the IDE user interface (e.g. file-chooser, message boxes). The refactorings itself are available after compilation in the menu bar or via context menus in the IDE itself.

Also, mbeddr and MPS clearly supported us in installing an ubiquitous language (`http://martinfowler.com/bliki/UbiquitousLanguage.html`) and improving separation of concerns.

6 What is the status?

The test language is not only used by developers, but also by test engineers. They can quite easily formulate tests and generate executable C code out of those descriptions. The Excel-based solution is not used anymore. Test-Engineers are responsible for developing test, a developer is responsible for the code-generator to make tests executable.