Functionality and Design of the CMock framework

Simon Raffeiner
Leitwerk AG, Appenweier
Baden-Württemberg, Germany
sraffeiner@leitwerk.de, sraffeiner@stud.fh-offenburg.de

Abstract*—Development cycles in the embedded world have not changed fundamentally for many years now. Even though the principles of agility, test-driven development and extreme programming have been adapted to embedded development by James Grenning [Gre02], Micah Dowty [Dow04], Michael J. Karlesky [KBE06] and others in the last ten years, the advantages of unit testing and mocking are widely ignored. The problem mainly arises from the misconception among developers that code written for embedded platforms without an operating system is hard to test because of the missing interaction possibilities with the system, and that space constraints make the use of frameworks impossible. Most developers focus on system testing instead. This paper shows how the CMock[1] mocking framework can be used in conjunction with the Unity[2] unit test framework to implement White-Box-Tests for embedded system software written in the C programming language.

I. Introduction

It is assumed that the reader is familiar with the terms "white-box testing" and "unit testing" in regards to software development.
Section 2 describes the fundamental problems developers are faced with when testing embedded systems. Section 3 gives a brief history of the CMock framework. Sections 4 describe the functionality, section 5 focuses on the internal design and integration with Unity. Section 6 gives general usage instruction, Section 7 showcases integration into existing build systems. Section 8 analyzes resource usage on different platforms (embedded and PC). The paper concludes with section 8, a brief overview of the limitations and an outlook into the future.

II. Testing embedded systems

Embedded systems are fundamentally different from Personal Computers in regard to available memory, processing speed and accessibility for debugging purposes. If unit testing is to be used one a large scale - preferably for all non-trivial functionality in every non-trivial module - the execution of tests must be tightly integrated into the tool-chain, run as fast as possible and deliver results that are as close as possible to the finally released system. Common solutions are found in the following list:

- A development board with a debugging interface that allows the automated download and execution of test binaries. Results are then transmitted to the host via an adequate interface (e.g. RS-232) or written to a fixed memory location which is read back via the debugging interface. This method is the slowest, but also the most accurate one when peripheral devices (timers, counters etc.) have to be tested.
- An emulator on the development host. This method is usually much faster than using a development board, but the emulator has to be scriptable and offer a debugging interface (which not all do). Most emulators are also not able to emulate peripheral devices at all or the emulation is not completely accurate.
- Cross-compiled tests for the development host architecture which are then executed locally. Developers have to pay attention to use compatible data types on both platforms (e.g. a standard integer in C has different sizes on different architectures). This method is the fastest and easiest one, but is not able to test peripheral devices or run native assembly code. Results may not be the same as when run on "real" hardware. Cross-compiling can however be used to easily detect errors in architecture- and hardware-agnostic decision logic.

The primary difference between embedded systems and workstations with complex graphical user interfaces is the absence of a separate presentation layer (View): Most embedded systems do not require user interaction, and if they do offer input and output through switches, sensors, LEDs, LCDs etc. signals are usually handled like any other form of I/O - directly through hardware drivers.

[1] http://cmock.sourceforge.net
[2] http://embunity.sourceforge.net

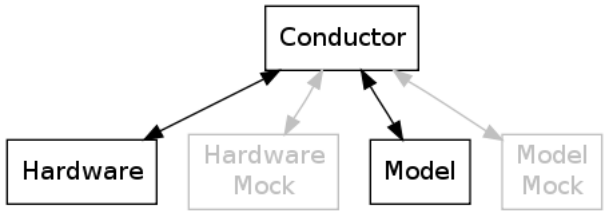

Fig. 1. Components in the Model-Conductor-Hardware pattern

[KBE06] introduces the "Model-Conductor-Hardware" (MCH) pattern for embedded systems development. It is basically an adaption of the well-known "Model-View-Controller" [BM00] pattern for systems without a dedicated user interface layer. MCH dictates that to get the most out of unit tests all software has to be split up into loosely coupled modules and each module is only allowed to consist of sub-modules implementing a single type of functionality: interface the hardware (Driver/Hardware), store data (Model) or realize decision logic (Conductor/Controller). Modules and sub-modules communicate through interfaces (in C terms "methods" listed in common header files), once an interface is established all code is written against this specification. This greatly enhances testability and re-usability as modules can be replaced and re-used as long as they fulfill the interface.

Because loose coupling would be senseless if unit tests then relied on the inner workings of specific modules mocking is used. Instead of handing full-blown instances of every needed module to a module under test only a minimal implementation fulfilling the interface is provided, decoupling the unit test from the need for a real implementation. Those "fake" modules can be generated automatically from the interface description (the header file), the expected behavior of the interface methods is simulated by "training" them with known values.

It has been proven that the usage of modules with well-defined interfaces improves overall code quality and enhances re-usability for other projects. [BM00] also shows that the concept comes with little to no overhead if an optimizing C compiler is used.

III. History

The basics of CMock have been developed by Atomic Objects, LLC[3] during a project for Savant Automation[4] [FBKW07], a major US-American manufacturer of automated guided vehicles.

[3]http://www.atomicobject.com

[4]http://www.savantautomation.com

Savant hired Atomic Objects in late 2005 to rewrite the firmware for two ARM9[5]-based electronic boards used inside the vehicles. Relying on an agile[6] development process, the team found itself in need of an embedded development tool-chain offering all functionality they had been used to when developing desktop or web applications, but at the same time fitting into a Microchip PIC[7] microcontroller with just 256 bytes of RAM and 32 kilobytes of ROM.

Atomic object successfully completed the project, adapting all agile development principles (unit testing, mocking and system tests) to the embedded environment and introducing new strategies (e.g. the Model-Conductor-Hardware pattern [KBE06]), special system test hardware and custom-written frameworks (namely Argent, Unity and CMock).

While it took the team nine months to finish the firmware for the first board, the second board (of comparable complexity) was completed in just four months by further tweaking the development process.

Two parts of the tool-chain, the unit test framework Unity and the mock framework CMock, were released as open-source projects to the public alongside a presentation [WV08] at the Embedded Systems Conference Boston in October 2008. Atomic Object employees Greg Williams, Michael Karlesky and Mark VanderVoord have since continued to improve both tool-kits and updated the package version from 1.0 to 1.2.2.

IV. Functionality

The CMock framework consists of a series of scripts written in the Ruby[8] scripting language and generates mock object code from C Header files containing function prototypes. All mocking code is output as standard C code.

CMock aims at mock testing only and therefore offers no unit testing functionality, which usually requires developers to use an additional unit testing facility. The framework comes with generator modules for Unity- or CException-style[9] methods.

A. Distribution

This paper relies on CMock in version 1.2.2 released in late 2008. The distribution obtained from the website contains the following directories:

- **config** Configuration files

[5]http://www.arm.com, the dominant processor architecture in mobile applications

[6]http://www.agilealliance.org

[7]http://www.microchip.com

[8]http://www.ruby.org, an object-oriented scripting language

[9]http://apps.sourceforge.net/mediawiki/cexception/, a simple exception handling mechanism written in ANSI C

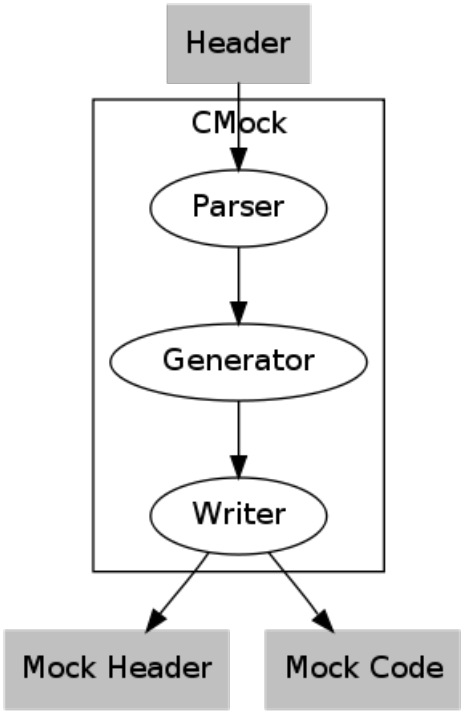

Fig. 2. Data flow inside CMock

- **docs** Documentation
- **examples** Example code
- **iar** Files specific to IAR Systems' Embedded Workbench Compiler for ARM architectures
- **lib** The actual framework
- **test** Unit tests for CMock itself
- **vendor** Needed additional libraries for the Unit tests

When CMock is included into a tool-chain the Ruby scripts from the lib/ sub-directory and the Unity framework from the vendor/ directory are necessary.

V. Internal Design

CMock follows the internal workings of every mock framework: offer some functionality to express how often, with which parameters and yielding which return value a method shall be called, store the information internally and fake the relevant method. If all values are inside specifications return the given value, if not fail.
Because the C language does not offer code manipulation at run-time (unlike Java and .NET) and CMock is targeted at embedded systems all code has to be generated before compilation. Mocks are generated in three steps (also see Figure 2):

- The parsing layer parses all header files and extracts information
- Extracted information is passed to the generator layer, which generates mock code using plug-ins
- The file writer layer updates or generates the necessary header and source files

Figure 3 shows the dependencies between the internal CMock modules. Two additional modules store global configuration options (config) and provide a list of pre-generated C template methods to the generator (generator_utils).

A. The parsing layer

The parsing layer consists of a single file, cmock_header_parser.rb. It parses a C header file and outputs the following information as a Ruby Hash structure in CMock-internal format:

- Included header files (may define new data types)
- External variables
- Function names, modifiers (e.g. static), return data types, argument names and data types

Figure 4 shows an example header file for a temperature filter module (obtained from the CMock distribution), Figure 5 the extracted information in its internal data structure.

Currently the parser works by first removing every line of input that does not contain needed information (comments, pre-processor directives, typedefs, defines etc.). It then continues to search for included header files and external variables and removes matching lines as well. Finally, all remaining lines are checked for prototype declarations and all names and data types extracted.
Matching and parsing is currently performed via regular expressions. Processing seems quite stable, but a proper C parser could improve quality and stability. As the parsing layer does not use any form of plug-in mechanism the usage of CMock for programming languages other than C would only be possible by replacing the current parser.

B. The generator layer

Mock code generation is split between the basic generator code in the file cmock_generator.rb and a list of generator modules (cmock_generator_*.rb) managed by the plugin manager (cmock_plugin_manager.rb).
The basic code handles the following tasks:

- Communication with the parsing layer (using the internal Hash structure)
- Communication with the file writer layer
- Generation of the mock header file including all needed external includes and external variables

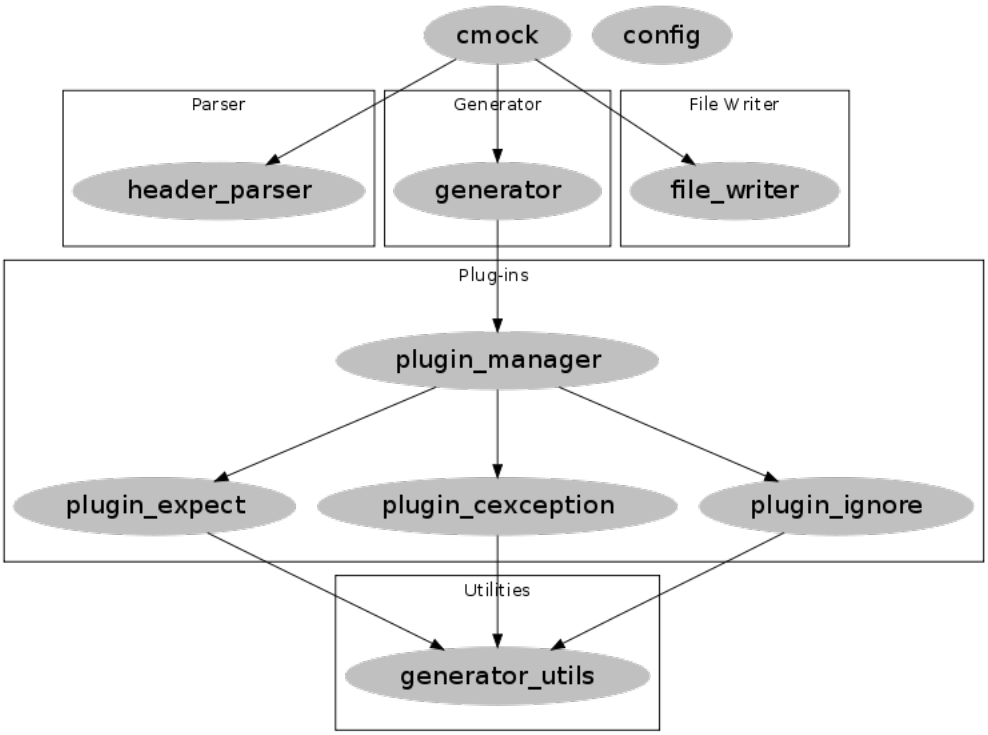

Fig. 3. Internal Dependencies between modules

```
#ifndef _TEMPERATUREFILTER_H
#define _TEMPERATUREFILTER_H
#include "Types.h"

void TemperatureFilter_Init(void);
float TemperatureFilter_GetTemperatureInCelcius(void);
float TemperatureFilter_ProcessInput(float temperature);
#endif // _TEMPERATUREFILTER_H
```

Fig. 4. TemperatureCalculator.h: Example Header file for a temperature filter module

- Generation of the global *_Init(), *_Destroy() and *_Verify() methods
- Generation of the mock instance structure

The generation of the mock methods is realized by plug-ins to allow better integration with unit test frameworks. Figure 6 shows the generated mock header file for the temperature filter, Figure 7 the generated mock instance object. The mock functions themselves are not included in the mock header file but are defined by including the original header file, in this case “TemperatureFilter.h”.
The mock instance object is the core data structure of every mock. It stores the following information:

- allocFailure, the total count of memory allocation errors (if any)
- *_Return_CallCount (actual number of calls) and *_Return_CallsExpected (number of expected calls) for every mock method
- *_Return (next), Return_Head (start) and Return_HeadTail (end), pointers into an array containing all registered return values for a mocked method
- *_$variable (next), *_$variable_Head (start), *_$variable_HeadTail (end), pointers into an array containing all expected values for a specific variable of a mocked method

The generated methods take care of the mock instance data structure. *_Init() resets all counters and arrays, *_Destroy() frees all memory allocated for arrays, *_Verify() compares call counter values to the expected count using methods provided by the used unit test framework.
_Expect(parameters) and *_ExpectAndReturn (parameters, return) work like in any other mocking framework: they increase the expected call count, store the expected arguments into the mock instance structure and, in case of *_ExpectAndReturn(), add the expected return value to the proper array.

```
{:includes=>["Types.h"], :externs=>[], :functions=>[{:modifier=>"", :
   rettype=>"void", :args=>[], :var_arg=>nil, :name=>"
   TemperatureFilter_Init", :args_string=>"void"}, {:modifier=>"", :
   rettype=>"float", :args=>[], :var_arg=>nil, :name=>"
   TemperatureFilter_GetTemperatureInCelcius", :args_string=>"void"}, {:
   modifier=>"", :rettype=>"float", :args=>[{:type=>"float", :name=>"
   temperature"}], :var_arg=>nil, :name=>"TemperatureFilter_ProcessInput"
   , :args_string=>"float␣temperature"}]}
```

Fig. 5. Generated internal Ruby Hash structure

```
#ifndef _MOCKTEMPERATUREFILTER_H
#define _MOCKTEMPERATUREFILTER_H
#include "TemperatureFilter.h"

void MockTemperatureFilter_Init(void);
void MockTemperatureFilter_Destroy(void);
void MockTemperatureFilter_Verify(void);

void TemperatureFilter_Init_Expect(void);
void TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(float
   toReturn);
void TemperatureFilter_ProcessInput_ExpectAndReturn(float temperature,
   float toReturn);
#endif
```

Fig. 6. MockTemperatureCalculator.h: Generated mock header file

```
static struct MockTemperatureFilterInstance
{
  unsigned char allocFailure;
  unsigned short TemperatureFilter_Init_CallCount;
  unsigned short TemperatureFilter_Init_CallsExpected;
  unsigned short TemperatureFilter_GetTemperatureInCelcius_CallCount;
  unsigned short TemperatureFilter_GetTemperatureInCelcius_CallsExpected;
  float *TemperatureFilter_GetTemperatureInCelcius_Return;
  float *TemperatureFilter_GetTemperatureInCelcius_Return_Head;
  float *TemperatureFilter_GetTemperatureInCelcius_Return_HeadTail;
  unsigned short TemperatureFilter_ProcessInput_CallCount;
  unsigned short TemperatureFilter_ProcessInput_CallsExpected;
  float *TemperatureFilter_ProcessInput_Return;
  float *TemperatureFilter_ProcessInput_Return_Head;
  float *TemperatureFilter_ProcessInput_Return_HeadTail;
  float *TemperatureFilter_ProcessInput_Expected_temperature;
  float *TemperatureFilter_ProcessInput_Expected_temperature_Head;
  float *TemperatureFilter_ProcessInput_Expected_temperature_HeadTail;
} Mock;
```

Fig. 7. Excerpt from MockTemperatureCalculator.c: Generated mock instance object

```
void testGetFormattedTemperature(void)
{
  TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(25.0f);
  TEST_ASSERT_EQUAL_STRING("25.0␣C\n", UsartModel_GetFormattedTemperature
     ());

  TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(-INFINITY);
  TEST_ASSERT_EQUAL_STRING("Temperature␣sensor␣failure!\n",
     UsartModel_GetFormattedTemperature());
}
```

Fig. 8. Usage of mocked modules with Unity

The generated mock methods compare the actual parameters to the expected parameters, maintain the internal data structures (e.g. increasing all pointers for the next call) and finally return an eventual value. If the whole call was unexpected or a parameter has an unregistered value the method fails using the methods offered by the unit test framework.
It should be noted that it is the responsibility of the tool-chain to correctly link test drivers and mock code together. Mocked methods have exactly the same signature as their corresponding "real" methods, if both are linked into a binary the linker is likely to complain about duplicate symbols.

C. The file writer layer

The file writer layer offers very limited functionality: Write output into a temporary file and overwrite/create a file with the given file name in a directory specified by configuration. As there is little complexity in this layer the reader is redirected to the cmock_file_writer.rb file for further analysis.

D. Integration with Unity

CMock is tightly integrated with the Unity framework. Although code generation should be framework-agnostic through the usage of plug-ins, it is not. Parts of the code are always automatically generated using Unity-style assertions.

Figure 8 gives an example on how a Model-Conductor-Hardware test driver method may use the generated temperature filter mock: The Usart module in its entirety communicates status information to a display unit. The Usart model inside the module gathers needed data and formats it according to specific rules. The test instructs the mock to return given values, in this case 25 and -INFINITY degrees (representing a sensor failure), and checks the resulting strings using Unity assertions. The tool-chain is responsible for linking the mock code against the test driver instead of the real code, the Usart model functions will not notice any difference and work with the mock.

VI. Usage

CMock will usually be included into some form of tool-chain and/or build system to automatically generate mock objects. It offers two different modes of operation: the cmock.rb script can be run through the Ruby interpreter on the command line, but it also exposes a class and some methods to generate mocks when included into other Ruby scripts. Both modes are equal in functionality, internally the command line mode just creates an instance of the class and executes the needed methods.

```
ruby lib/cmock.rb -oconfig.yml src/
   uart.h src/sensor.h
```

Fig. 9. Running CMock from the command line

```
cmock = CMock.new(options)
cmock.setup_mocks(list)
cmock.generate_mock(source)
```

Fig. 10. Ruby functions offered by CMock

A. Mocking from the command line

When executed directly from the command line all parameters are expected to be path-names to C header files or an optional configuration file. CMock will then iterate through all file names and generate mock code. Figure 9 gives an example on how CMock may be run from the command line.

Configuration files are expected to be written in YAML[10] format. A full list of currently supported configuration options may be found in the project documentation.
If no configuration file is specified the defaults are used, which at this moment activates the cexception and ignore plug-ins and places all generated mock code into a sub-directory called mocks/.

B. Mocking from Ruby scripts or via Rake

CMock may be incorporated as a native Ruby object as shown in Figure 10. It only offers a constructor and two simple methods.

The constructor accepts the same options as the ones specified in a YAML file when running CMock from the command line. If nothing is specified the defaults are used. `setup_mocks()` generates mock objects for a list of input files, `generate_mock()` for a single file.
The cmock.rb file just instantiates a CMock object and passes all command line parameters to the `setup_mocks()` method, it is therefore very easy to include CMock into an existing tool-chain if it already uses Ruby scripts or the Rake[11] build system.

VII. Integration into existing projects

The CMock distribution and examples rely on the Rake build system, but integration into pre-existing build systems is quite simple: Create an additional build target that calls on Ruby and cmock.rb to generate the mocks before actual compilation

[10] http://www.yaml.org/, YAML Ain't Markup Language, a human-friendly data serialization standard

[11] http://rake.rubyforge.org/, drop-in replacement for make

```
cmock:
  mock_path: 'mocks/'
  includes:
    - 'Types.h'
  plugins:
    - 'ignore'
```

Fig. 11. An example YAML configuration

	Code	Instance
Atmel AVR (Atmega8)	1664	31
Atmel AVR (Atmega32)	1714	31
Intel 8051	1681	31
Intel i386	1775	60
AMD x86_64	1804	104

Fig. 12. CMock resource usage

happens. The wiki gives an example[12] on integration into the popular Eclipse IDE[13].

Most projects will have a source tree with pre-existing folder structure and at least in the beginning the few test drivers and mocks usually "live" in one folder with "normal" source files. This is not the best option, as IDEs with extended functionality like indexing and code completion will initially fail - suddenly many methods appear as duplicates (once for the mock and once for the original source file) and additional auto-generated mock methods (Init(), Destroy(), *_Expect() etc.) start appearing. It may be possible to exclude single files from indexing but it this process is error-prone and generates much more problems than simply putting mocks, source files and tests into different folders right from the beginning. In this case all unwanted folders just have to be removed from the indexer once.

All CMock parameters, including the output path for mock files, are easily configurable through the usage of YAML files. An example is shown in figure 11: It tells the generator layer to include a needed header file, Types.h, into all mock code, and load the "ignore" plug-in. The file writer layer is advised to put all generated files into the "mocks" sub-folder of the current working directory.

A complex build setup may require multiple YAML files containing different options and paths.

VIII. Resource usage

Most embedded systems are limited in code storage (internal/external Flash or EEPROM) and especially memory (internal/external RAM). If unit tests are to be run in-situ or in an emulator their space consumption should be as low as possible so many tests can be run out of a single binary, speeding up the process and reducing the number of build targets. The first column in Table 12 compares the total code size of all mock methods generated for the temperature filter example introduced in Figure 4, compiled on different platforms. The second column compares the size of the generated mock instance object on the same platforms, respectively. These numbers do not change at run-time.
What changes however is the memory space needed for expected arguments and return values: Every call to *_Expect(parameters) or *_ExpectandReturn (parameters, return) increases memory usage at run-time. The exact numbers are subject to the size of exact data types and the number of values to store. Also the specific implementation of malloc() on the given platform may not be able to use every single byte of memory but rather allocate aligned blocks.
Compilers used: avr-gcc 4.3.2 for Atmel AVR, SDCC 2.8.0 #5117 for Intel 8051, GCC 4.3.3 for Intel i386, GCC 4.3.3 for AMD x86_64. All compilers have been instructed to produce the smallest possible binary via the according command line flags.

As the table shows generated code size is very similar when compared between platforms, but the impact can be dramatic when compared to the limitations of a specific device: The Atmega8 e.g. offers only 8 kilobytes flash storage and 1 kilobyte of RAM. This is enough for projects of a considerable size (people have built complete MP3 players using it), but we could probably just fit two or three mocks into its memory - the rest is needed for the actual unit tests and other subsystems (host communication etc.). Some members of the Intel 8051 family come with only 128 or 256 bytes of RAM, mocks may be useless on such a platform because there is not enough memory to store expected parameters and return values (depending on the rest of the test setup).
The situation is however considerably better on the Atmega32 (32 kilobytes flash, 2 kilobytes RAM) and many other platforms (ARM, MIPS, ColdFire etc.). Intel i386 and AMD x86_64 architectures are mainly listed as examples for cross-unit testing on a development host machine where resource limitations are usually not an issue.

[12]http://sourceforge.net/apps/trac/cmock/wiki/EclipseIde, Eclipse IDE Integration

[13]Integrated Development Environment

IX. Limitations and Conclusion

It has been proven that CMock offers the basic functionality needed to mock standard C methods. There are however some unresolved issues that may lower its usability for embedded system development.

CMock makes heavy usage of the malloc(), realloc() and free() standard library calls. Not every runtime environment offers memory management, some embedded coding style policies may even prohibit the usage of malloc.h - in test code and in cross-compiled code for development workstations as well. Currently there is no solution for this problem.

CMock is heavily tied to the Unity unit test framework released by the same company and the Rake build system. Although code generation should be fully handled by plug-ins parts of the generated mock code will always rely on Unity methods for assertions and failures. Integration with existing development tool-chains may be impossible under this circumstances.

The CMock framework is a very young project and has been actively developed since its release. The Subversion code repository on sourceforge.net shows constant activity. Time will tell if CMock has the potential to become a worthy addition for embedded tool-chains or if the developers focus too much on their own needs and the strong dependencies on Unity and Rake pose a threat to its success.

As there is currently no hype about unit testing and mocking in the embedded world it may take some time until enough developers from other projects have evaluated CMock for their needs.

References

[BM00] Andy Bower and Blair McGlashan, editors. *Twisting The Triad - The evolution of the Dolphin Smalltalk MVP application framework.* ESUG 2000 International Smalltalk Conference, 2000.

[Dow04] Micah Dowty. Test driven development of embedded systems. University of Colorado at Boulder, March 2004.

[FBKW07] Matt Flettcher, William Bereza, Mike Karlesky, and Greg Williams, editors. *Evolving into Embedded Development*. Agile 2007 Conference, August 2007.

[Gre02] James W. Grenning, editor. *XP and Embedded Systems development*. Object Mentor Inc., 5101 Washington Street, Suite 1108, Gurnee, IL 60031 USA, 1 edition, March 2002.

[KBE06] Michael J. Karlesky, William I. Bereza, and Carl B. Erickson, editors. *Effective Test Driven Development for Embedded Software*. IEEE Electro/Information Technology Conference, 2006.

[WV08] Greg Williams and Mark VanderVoord, editors. *Embedded Feature Driven Design Using TDD and Mocks*. Embedded Systems Conference Boston 2008, October 2008.