Embedded Software Development by means of Remote Prototyping

Piet Cordemans, Sille Van Landschoot and Jeroen Boydens

Abstract – Modern software development strategies emphasize on iterative development, because shorter cycles deliver feedback on quality more quickly. However programming embedded software is considered too slow for this process. A technique called remote prototyping delegates hardware dependent calls to the effective target, while software is developed on host. A proof of concept was developed, which delivered first indications of added value of the technique. Furthermore a critical reflection is provided, which compares remote prototyping with an alternative technique, called dual targeting.

Keywords – Embedded software development, inter-process communication

I. PROBLEM STATEMENT

Currently, debugging software, managing code complexity and the development process are indicated as current major concerns for embedded systems [6, 7]. Next, as indicated by Boehm [2] the cost related to fixing a software bug is exponentially correlated with the time needed of finding it. Consequently detecting the bug early on is essential to keep costs under control. Fundamental to a timely identification of problems is acquiring feedback on the quality of software shortly after developing it [3].

Thus far, iterative developing on the embedded target system is considered too slow, since software needs to be migrated off the development environment, i.e. the host system. Uploading code to the target program memory does not only slow down embedded software development, but also delays obtaining the necessary feedback to detect bugs early on.

Cross-platform development deals with this problem, as large parts of embedded software can be developed off-target. However, this requires a thorough hardware abstraction, which in turn is susceptible to bugs. Moreover, the effort required to provide a sufficient abstraction can lead to a high cost, which can be considered as pure overhead. Stable platforms with a large number of application developers, such as mobile phones, justify extensive target emulation on the host system. Yet, a lot of embedded systems tend to be focused and dedicated. Furthermore only a handful of developers are involved.

Therefore these systems cannot justify the cost of building dedicated emulation environments.

These problems require a solution which minimizes effort put in hardware abstraction, while obtaining a high accuracy of hardware representation in the development environment. Furthermore, in order to avoid slowing down the development process, developing on a host system is designated.

Considering this set of conditions remote prototyping is proposed. Remote prototyping allows developing embedded software on host, while hardware involvement is delegated to the embedded system under construction.

First, the development process with remote prototyping is discussed. Consequently the results of developing three proof of concept applications are shown. Next, a critical reflection is provided in which embedded constraints, issues inherent to remote prototyping and the dependency with tests are discussed. Finally future work on the subject and a conclusion is given.

II. REMOTE PROTOTYPING

In order to obtain a fast development cycle, which provides accurate feedback, the approach of remote prototyping has been developed. First the general principle is clarified and afterwards an in-depth architectural discussion is given.

A. Principle

As illustrated in figure 1, remote prototyping involves developing code on the host platform, while specific hardware calls can be delegated towards the respective code on target. Addressing the hardware-related functions on host, delegating the call to the target and returning values as provided by the function prototype are provided by Inter-Process Communication (IPC) infrastructure.

Software can be developed on host, as all hardware functionality is provided in the form of function stubs. These stubs deliver the function definition on the host system while an effective call to the function stub will delegate the call to the target implementation. Stub definitions are provided by the broker on host. Furthermore the broker also provides communication with its counterpart on target. The broker on target keeps a list of addressable functions and also retains a reference of statically and dynamically allocated memory chunks for future addressing. Code on target, which is instrumented to be remotely addressable, is called a skeleton.

P. Cordemans, S. Van Landschoot and J. Boydens are with the Department of Industrial Sciences and Technology, KU Leuven, University College – Zeeijk 101, B-8400 Oostende, Belgium, e-mail: (Piet.Cordemans,Sille. VanLandschoot,Jeroen.Boydens) @kbo.be

J. Boydens is with the Department of Computer Science, K.U.Leuven – Celestijnenlaan 200A, B-3001 Leuven, Belgium, e-mail: Jeroen.Boydens@cs.kuleuven.be

Research funded by IWT-090191
Remote prototyping is effective under the assumption that software under development is evolving, but once the software has been thoroughly tested, a stable state is reached. As soon as this is the case the code base can be instrumented to be remotely addressable. Subsequently it is programmed into the target system and thoroughly tested again to detect cross-compilation issues. Once these issues have been solved, the new code on target can be remotely addressed with the aim of continuing development on the host system.

The main difference between developing and prototyping software in the host environment is their affiliation with tests. Conventionally software is merely developed on the host system and only debugged afterwards with ad hoc methods. Namely once it has been ported to the target system. This typically infers an unacceptable delay in a fast iterative process. However remote prototyping allows testing software before it has been ported, regardless whether tests are written before code is developed, following the principle of Test-Driven Development [1], or afterwards in a more conventional fashion.

B. Process

An overview of the remote prototyping process applied to an object oriented implementation, for instance C++, is given in figure 2. A fundamental difference exists when all objects can be statically allocated or whether dynamic creation of memory objects is required.

In a configuration in which the target environment can be statically created, setup of the target system can be executed at compile time. The broker system is not involved in constructing the required objects, yet keeps a reference to the statically created objects. Effectively the host system does not need to configure the target system and treats it as a black box.

Conversely the process of remote prototyping with dynamic allocation requires additional configuration. Therefore the target system is approached as a glass box system. This incurs an additional overhead for managing the on target components, yet allows dynamically reconfiguring the target system without wasting a program upload cycle.

The dynamical remote prototyping strategy starts with initializing both the broker as well on target as on host side. Next, a test is executed, which initializes the environment. This involves setting up the desired initial state on the target environment. This is in anticipation of the calls, which the software under development will conduct. For instance, to create an object in the target, the following steps are performed, as illustrated in figure 2.

1) The test will call the stub constructor, which provides the same interface as the actual class.
2) The stub delegates the call to the broker on host.
3) The broker on host translates the constructor call in a platform independent command and transmits it to the target broker.
4) The broker on target interprets the command and calls the constructor of the respective skeleton and in the meanwhile assigns an ID to the skeleton reference.
5) This ID is transmitted in an acknowledge message to the broker on host, which assigns the ID to the stub object.

Ensuing test setup, the test is effectively executed. Any calls to the hardware are dealt with by the stub object, which are delegated to the effective code on target. Likewise, any return values are delivered to the stub. Optionally another test run can be done without rebooting the target system. A cleanup phase is in order after each test has executed, otherwise the embedded system would eventually run out of memory. Deleting objects on target is as transparent as on host, with the addition that the stub must be cleaned up as well.

III. RESULTS

Remote prototyping has been applied in three different case studies, which were part of a comparative research between strategies to apply Test-Driven Development to embedded software [4]. First, a proof of concept application was developed for a simple digital temperature sensor. Next, the broker infrastructure was ported to a loom control system starting from a considerable amount of legacy code. The purpose was to rewrite part of an application according to the remote prototyping principle. Finally, an alternative to the in-house developed broker infrastructure was explored. Namely, the RPC library of NXP[9] was used as the IPC infrastructure of choice.

The in-house developed broker on target infrastructure is lightweight, 192 executable logical lines of code. It is also portable, only depending on two common C++ standard library classes, this is stringstream and string. The infrastructure requires 20 executable logical lines of code to make a class remotely available, adding two extra lines per method.
As the NXP RPC library is closed source, no metrics are available.

The on-host IPC infrastructure was developed using the .Net framework and Microsoft Visual Studio. Unit testing framework.

**IV. Reflection**

Remote prototyping deals with certain constraints inherent to embedded systems. However, some issues were encountered when implementing and using the IPC infrastructure. Finally, a reflection is given on how tests and the fast feedback cycle they enable will influence code quality.

**A. Constraints**

The impact, especially considering constrained memory footprint and processing power, of the IPC infrastructure on the embedded system is minimal, as shown in the results section. Of course it introduces some overhead to systems which do not need to incorporate the infrastructure for application needs. On the other hand, remote prototyping enables conducting unit tests with a real target reference. Porting a unit test framework and running the tests in target memory as an alternative will introduce a larger overhead than the IPC infrastructure and lead to unacceptable delays in an iterative development process.

Next, the embedded infrastructure does not always provide all conventional communication peripherals, for instance Ethernet, which could limit remote prototyping applicability. However, since a common communication format is used, the effective communication layer is abstracted. Moreover, the minimal specifications needed to setup remote prototyping are limited as throughput is small and no timing constraints need to be met.

Finally, remote prototyping requires that hardware and a minimalistic hardware interfacing is available. This could be an issue when hardware still needs to be developed. Furthermore hardware could be unavailable or deploying code still under development might be potentially dangerous. Lastly, a minimalistic software interface wrapping hardware interaction and implementing the IPC infrastructure is needed to enable remote prototyping. This implies that it is impossible to develop all firmware according to this principle.

**B. Issues**

The encountered issues when implementing and using remote prototyping can be classified in three types. First are cross-platform issues related to the heterogeneous architecture. A second concern arises when dynamic memory allocation on the target side is considered. Thirdly, translation of function calls to common architectural independent commands introduces additional issues.

Differences between host and target platform can lead to erratic behavior, such as unexpected overflows or data misrepresentation. These issues could be introduced such as word size, endian ordering or floating point representations. However, most test cases will quickly detect any data misrepresentation issues. Likewise, border problems can be discovered by introducing some boundary condition tests.

Next, on-target memory management is an additional consideration which is a side-effect of remote prototyping. Considering the limited memory available on target and the single instantiation of most driver components, dynamic memory allocation is not desired in embedded software. Yet, remote prototyping requires dynamic memory allocation to allow flexible usage of the target system. This introduces the responsibility to manage memory, namely creation, deletion and avoiding fragmentation. By all means this only affects the development process and unit verification of the system, as in production this flexibility is no longer required.

Finally, timing information between target and host is lost because of the asynchronous communication system, which can be troublesome when dealing with a real-time application. Furthermore to unburden the communication channel, exchanging simple data types are preferred over serializing complex data.

**C. Tests**

The purpose of remote prototyping is to introduce a fast feedback cycle in the development of embedded software. Introducing tests, which deliver the desired feedback, has some consequences. First, tests allow verification of functional specifications, which is not possible with static analysis. Also, tests can identify execution differences between the host and target platform. In order to do so the code under test needs to be ported from the host system to the target system. By instrumenting code under test, the remote prototyping infrastructure can be reused to execute the tests on host, while delegating the effective calls to the code on target.

Similarly to tests, remote prototyping has a profound effect on the design of software. In effect, it puts the focus on the interface of a software component, rather than its implementation. Moreover remote prototyping enforces the practice of separating low level drivers, from software components on a higher level of abstraction.

**V. Related Work**

Incremental embedded software development with a fast feedback system is already proposed by Grenning [5] for Test-Driven Development of embedded software. However his dual targeting principle differs fundamentally from remote prototyping. Dual targeting involves a build on host and a separate build on target. On host hardware calls are stubbed out with test doubles. These will mimic hardware behavior to the extent of running tests, which verify the functional specifications of the code under test. In order to verify the assumptions made on the hardware behavior, the developed code can be ported to the target alongside the unit testing framework and tests. Developing and managing test doubles requires an additional development overhead. However, contrary to remote prototyping, dual targeting
can be used to develop software when the embedded target is unavailable.

The LDRA tool suite[8] supports host/target testing, which encompasses the idea of instrumenting code on target to make it addressable by tests on the host system.

VI. FUTURE WORK

Remote prototyping has only been applied in proof of concept applications in which the boilerplate code necessary to setup IPC infrastructure was written by hand. Instrumenting code on target and producing the stub communication infrastructure on host, introduces an additional overhead in the development process. This could be eliminated or reduced to a minimum by automated code generation.

Moreover as real-time properties are lost in the asynchronous communication process between host and target, an important characteristic of embedded systems is ruled out of remote prototyping. Yet extending the communication protocol with time stamps obtained from timers on target, could retrieve at least a partial notification of real-time properties.

VII. CONCLUSION

In this paper an iterative embedded software development process, called remote prototyping, was described, which deals with some impeding embedded system properties, such as a slow upload cycle and memory limitations. Furthermore remote prototyping reduces overhead introduced into the development cycle to a minimum, while retaining feedback from the effective hardware for testing purposes.

Three case studies demonstrated how tooling can support the process, by either custom made or reusing existing IPC infrastructure.

Finally a reflection was given on remote prototyping considering the embedded system constraints, communication between heterogeneous platforms and quality assurance with tests.

REFERENCES