The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MSc in Software Engineering.

Model-driven software migration between microcontrollers

Sophie Wood

Version 0.1, 2018-May-16

Supervisor: Simos Gerasimou

Number of words = 0, as counted by `wc -w`.
This includes the body of the report, and Appendices TODO, but not TODO.

Abstract

TODO

Acknowledgements

TODO

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

Bartels et al. define obsolescence as "materials, parts, devices, software, services and processes that become non-procurable from their original manufacturer or supplier" [1]. Both software and hardware can be subject to obsolescence problems.

Issues with hardware obsolescence have been driven by the growth of the electronics industry. This has reduced the life cycle of electronic parts as competitors release products with better functionality and features. Existing products are no longer commercially viable and therefore go out of production [1]. This is a particular issue in the defence/aerospace sectors as the typical life cycle of a system is 20-30 years or longer [2]. As such, parts will become unavailable before the system is completed. Singh et al. found that these systems "often encounter obsolescence problems before they are fielded and always during their support life" [3].

Obsolescence is also a concern in the software industry. Businesses must continuously update their products in order to stay ahead. Bill Gates has said:

> "The only big companies that succeed will be those that obsolete their own products before someone else does." [1]

There are three main causes of software obsolescence: *logistical* - digital media obsolescence, formatting or degradation terminates or limits access to software; *functional* - hardware, requirements, or other software changes to the system obsolete the functionality of the software; and *technological* - the sales and/or support for commercial off the shelf (COTS) software terminates [4].

In addition, software obsolescence can be driven by hardware obsolescence and vice versa. Another issue is lack of skills. For example, there may be a limited number of people that are competent in the language the system is written in. If these people leave the company, the system can no longer be maintained [5].

Again avionics/military and other “safety-critical” systems particularly struggle with software obsolescence issues as even small changes may have to go through extensive and costly qualification/certification processes [3]. Consequently, software obsolescence costs can equal or exceed that of hardware [4]. Despite this, strategies for obsolescence mitigation/management have generally focused on hardware obsolescence problems. It is important that both hardware and software obsolescence issues are tackled as these problems can be incredibly costly to these industries. For example, the US Navy estimates that obsolescence problems can cost up to $750 million annually [6]. Sandborn and Myers also found that sustainment costs (which include costs related to obsolescence) dominate the system costs in the case of development of an F-16 military aircraft [7].

Current strategies for mitigating software obsolescence issues are insufficient. Proactive measures (e.g. improving code portability or using open-source software) often require either a large amount of resources or resources that are unavailable. Reactive strategies for tackling obsolescence (e.g. software license downgrades, source code purchase or third party support) may not always be possible. When these methods are inadequate, the legacy system may have to be redeveloped or rehosted [4]. However, many such software modernisation projects are abandoned or not completed within the planned timescale/budget [8].

If redevelopment projects can be automated or even partially automated, they can more easily be completed on time. [9] uses code analysis and code-based transformations to partially automate the process of adapting software to work with different libraries as well as migrating software between hardware platforms. Although this approach is successful, the process may not scale well. For example, in one stage of the process of adapting code to use a new library, an abstraction layer is generated that has the same usage behaviour as the obsolete library. This is then used to replace the usages of the legacy library. If applied to a project with many library usages, this could generate a large amount of additional code in the modernised project which may make it difficult to maintain.

An alternative approach is to use model-driven software modernisation (MDSM) for (partial) automation of modernisation projects. MDSM is based on the use of model-driven engineering (MDE) principles such as models and transformations to facilitate the migration process (MDE and its application to software modernisation are discussed in more detail in Sections 2.1.2 and 2.2). For example, model-to-model transformations can be used to map a model of the legacy code to a model of the new system. A model-to-text transformation can then be applied to this new model to generate code for the new version of the system.

MDSM is a promising approach as it has already been proved feasible in industry by the Sodifrance[1] company. For example, they were able to migrate a large-scale banking system (at a size of around one million lines of code) to J2EE [10]. Kowalczyk and Kwiecińska have also demonstrated the viability of MDSM by modernising a project written in Java 1.4 and the Hibernate 2.x framework into Java 1.5 and Hibernate 3.x [8]. One benefit of MDSM is that tools (e.g. model discoverers) can sometimes be reused between projects making subsequent modernisation projects quicker and cheaper to complete. For example, MoDisco is an open-source model-discoverer that can generate models from Java code [11].

The Defence Science and Technology Laboratory (DSTL) at the Ministry of Defence (MoD) have identified several instances of software obsolescence they would like to tackle. A problem of particular interest is "the migration of an entire software system from a legacy hardware platform to a modern more powerful platform" [9]. This report will explore how MDSM processes can be used to partially automate the migration of code between microprocessors. A case study implementing the migration of code between a Parallax Propeller Activity Board and an Arduino Uno will be used to demonstrate the approach developed.

1.2 Project Goals

TODO

1.3 Project Scope

TODO

1.4 Report Structure

TODO

[1]https://www.sodifrance.fr/

2 Literature Review

The following chapter introduces the existing work on software obsolescence and the technical background required to understand this project. It begins with an overview of the existing approaches for software obsolescence mitigation in Section 2.1.1 and in particular, MDE-based strategies in Section 2.1.2. The specific problem of migration between microcontrollers and the existing methods for facilitating this are discussed in Section 2.1.3 as well as how the MDSM-based method explored in this project is related.

Sections 2.2 and 2.3 contain the technical background of the project including a brief overview of MDE and the chosen framework for this project as well as a description of the microcontrollers used for the case study.

2.1 Existing Work

2.1.1 Approaches to Software Obsolescence

Rojo et al. identify some of the main methods of tackling obsolescence in their paper "Obsolescence management for long-life contracts: state of the art and future trends" [2]. Although they identified many strategies for handling the obsolescence of electronic components, very few methods were relevant for dealing with software obsolescence.

The tools "se-Fly Fisher" and "R2T2" (Rapid Response Technology Trade) were identified as useful for managing software obsolescence. These tools are used for design refresh planning. Design refresh planning deals with obsolescence in a proactive manner by planning the best times for performing a redesign of the system during the sustainment stage of its lifecycle.

Se-Fly Fisher uses the technology curves of each part of the system to: forecast how often a system baseline should change; identify replacement resources and estimate the benefit of each system baseline change [2].

Similarly, R2T2 can: forecast system obsolescence; allow for comparisons to alternative solutions; produce a life cycle obsolescence plan for the elements that require refreshment and plan when element replacements should occur [12].

These forecasting tools may enable organisations to plan ahead and handle cases where equivalent parts/software are available. However, they cannot assist in the actual redesign or redevelopment process which can be time consuming and costly.

Similarly, Sandborn et al. identified very few approaches towards managing software obsolescence. The main methods they identified for mitigating software obsolescence were [4]:

(1) Software License Downgrade: users can purchase licenses for the current product and apply them to older versions.

(2) Source Code Purchase: customers purchase the source code for the product.

(3) Third Party Support: a third party is contracted to maintain support for the software.

When possible, these approaches could reduce costs as the software does not have to be maintained in house or redeveloped. However, sometimes these methods may not be possible. Additionally, given that the system owners are so dependent on the legacy code, it puts them in a poor position for negotiating the prices for these contracts and so this approach could become costly.

2.1.2 MDE Approaches to Software Obsolescence

More recently, MDE approaches have been used to tackle software obsolescence problems where legacy code must be redeveloped. MDE allows some of the stages in migration to be automated, consequently reducing the timescale and costs involved.

The company Sodifrance has successfully been using MDE for development and migration projects for over ten years [10]. The general MDE approach to migration used by Sodifrance is shown in Figure 2.1. The approach is separated into four stages as follow:

1. Firstly, the code of the legacy application is parsed in order to create a model of the legacy system. In Figure 2.1, *L* indicates the metamodel for the implementation language of the legacy code.