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 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

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. summarise the existing literature for tackling obsolescence up to 2010 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. Yet, they propose that sectors such as the defence industry would benefit from managing this problem.

They did identify two tools "se-Fly Fisher" and "R2T2" (Rapid Response Technology Trade) that could be 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.

2. This model is then transformed to a Platform Independent Model (PIM) conforming to an ANT metamodel. This model represents a high-level view of the legacy code. This process is dependent on knowledge of the libraries and coding conventions used by the legacy platform.
3. Next the PIM has a model transformation applied to produce a Platform Specific Model (PSM) conforming to a UML metamodel. The high-level views from the PIM are adapted to fit the target platform.
4. Finally, code is generated from the PSM by using template-based text generation tools.

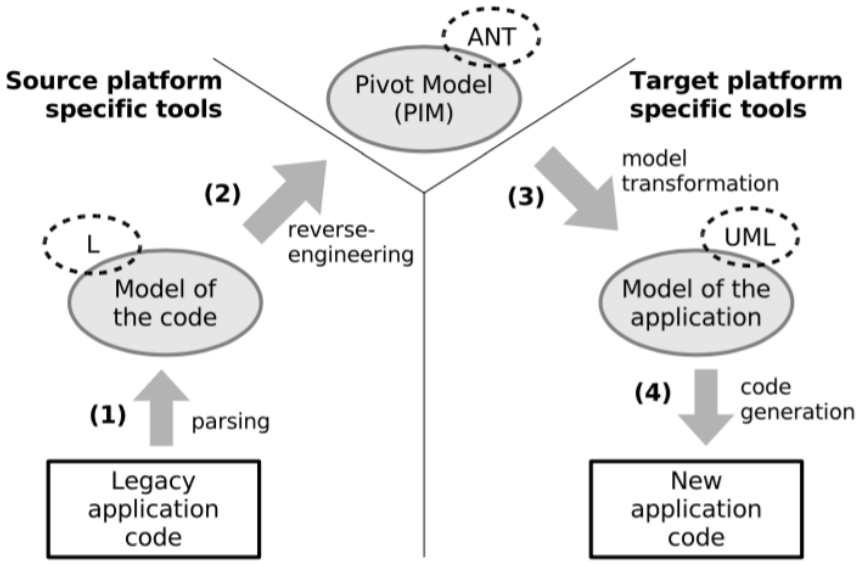

Figure 2.1: The general model-driven migration process used by Sodifrance [10].

The first two steps of the process can usually be fully automated. However, the remaining stages need some manual effort. Tasks that could not be completed automatically are indicated in the generated code (e.g. by TODO directives in Java applications) and summarised into

a task list in order to allow the manual process to be completed more efficiently.

Sodifrance demonstrated the effectiveness of their approach using a case study of migrating a large-scale banking system from Mainframe to J2EE.

Kowalczyk and Kwiecińska have also explored the use of MDE in software migration [8]. They identified two main approaches: the "Reverse MDA approach" (RMA) and the "Direct Transformation Approach" (DTA).

The RMA approach follows the same stages as that used by Sodifrance. The DTA approach shown in Figure 2.2 reduces the number of stages involved in the transformation process by directly transforming between platform specific models.

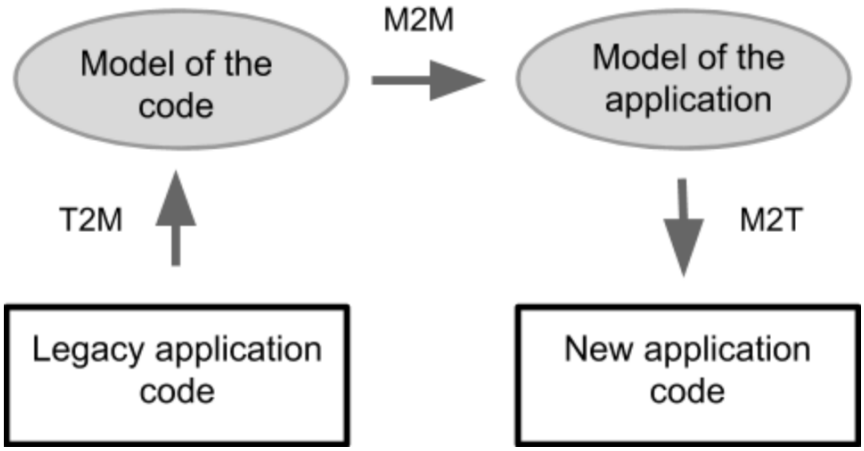

Figure 2.2: The stages involved in the "Direct Transformation Approach".

The feasibility of both approaches was demonstrated by a case study migrating from Java 1.4 and the Hibernate 2.x framework to Java 1.5 and Hibernate 3.x. Based on qualitative analysis of the two approaches, they recommend to use an approach similar to DTA as it is often not the case that a transformation from the legacy code model to the PIM will exist. It is especially suited to cases where the code model and application model use the same metamodel.

MDE approaches to software migration have many benefits. The

primary reason is the ability to automate some of the stages in the migration process, consequently reducing the cost and time scale of projects. Secondly, parts of the process (e.g. model discoverers or transformations) can be reused between different migration projects which will again cut the cost of migration.

There are however, drawbacks to the MDE approach. There is a high cost of entry to start using MDE methods — the initial development of processes and tools can be time consuming and expensive if open-source tools are not available. Secondly, a particular issue in commercial projects is that no code will be available until the initial analysis and tool development stage of the process is completed. In the case of the Sodifrance case study, code was not delivered until 10 months after the project had begun [10]. This can make customers nervous and reluctant to try this approach. On the other hand, once the initial stage is completed, the delivery rate of the code can be much faster than for a standard re-development approach.

One of the main considerations that must be made when choosing an MDE approach is the size of the project. If the project is small, it may be more expensive to use this method as the initial stages may take longer than manually migrating the code. This is demonstrated in Figure 2.3.

2.1.3 Migration Between Microcontrollers

As mentioned previously, a particular problem of interest to the MoD is "the migration of an entire software system from a legacy hardware platform to a modern more powerful platform" [9].

Atmel addressed the issue of migrating code between microcontrollers by reducing the learning curve for developers moving from development on 8-bit to 32-bit platforms. They aimed to achieve this by providing powerful debug facilities and development tools similar to established 8-bit development platforms [13].

Although this strategy should reduce the migration effort, the code will still be have to be migrated manually which is error-prone and difficult. Additionally, the development of new tools adds extra overhead to the project, although these tools are reusable between projects.

Another approach for migrating code between microprocessors combines code analysis, code-based transformations and verification/validation techniques [9]. The procedure follows the following steps:

(1) Software system analysis (automated): The source code and obsolete

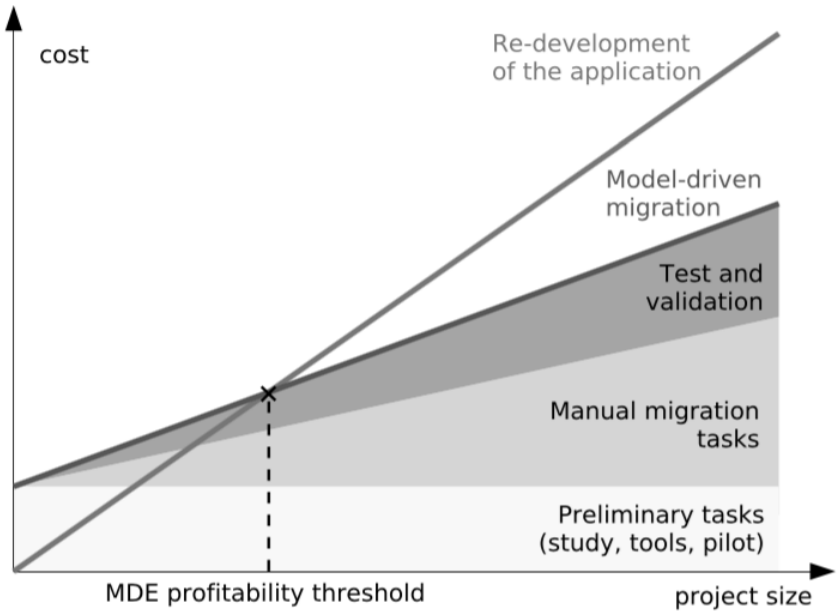

Figure 2.3: Project migration cost as a function of its size [10].

libraries are parsed to obtain abstract syntax trees (ASTs). The AST can be examined to indicate the elements using the obsolete library and help establish the system's dependency level.

(2) Discovery of similar libraries (non-automated): The development team identifies candidate libraries for replacement of the obsolete library.

(3) Compatibility analysis of the discovered libraries (non-automated): Candidate libraries are rejected if they don't conform to technical or semantic requirements.

(4) Data visualisation (automated): Used to analyse the software system and its coupling with the obsolete library.

(5) Execute generation transformations (automated): Firstly, an abstraction layer is generated that has the same usage behaviour as the obsolete library. Next the obsolete library usages are replaced with this abstract layer.

(6) Mappings inference (non-automated): A developer creates a list of mapping rules by inspection of the obsolete and replacement libraries.

(7) Code population (non-automated): A developer uses the mapping rules generated in stage 6 to populate the abstraction layer generated in stage 5.

This method was demonstrated to succeed in partially automating the migration of software between an Arduino and Raspberry Pi.

By automating some of the migration stages, the overall time and cost of software modernisation should be reduced. Another benefit is that the code analysis stages allow for potential risks to be detected early on. However, there is still a lot of manual effort required for migrating between libraries and in particular, generating mapping rules requires a developer with a strong understanding of both the new and obsolete libraries. Another issue is that adding the abstraction layer could lead to a large increase in the size of the code if the obsolete libraries are used frequently. This could make the code more difficult to maintain.

My project aims to explore an alternative approach for migrating code between microcontrollers by applying an MDE approach similar to that discussed in Section 2.1.2. In doing so I aim to be able to (partially) automate the migration process, consequently reducing the time taken for migration whilst also avoiding issues associated with other methods such as code blow-up in [9].

2.2 Model Driven Engineering

Model Driven Engineering (MDE) was developed in order to address complexity within the problem space of computing. In the past, approaches for enabling programmers to develop code more easily have focused on simplifying the solution space. This was achieved by providing abstractions such as higher-level programming languages or providing operating systems to manage the difficulty of programming hardware directly. However, these solutions are unable to express domain concepts effectively, unlike MDE techniques [14]. Furthermore, it has been found that MDE can also be utilised in automating the software development process [15].
The rest of this section introduces the terminology and main components of MDE as well as the MDE framework I will use in the project.

2.2.1 Domain-Specific Modelling Languages (DSMLs)

Meta-meta-modelling mechanisms such as the Object Management Group's (OMG) Meta Object Facility (MOF) can be used to create modelling languages for a given problem domain [16]. The key components of the OMG's approach to DSMLs are [17]:

Models: A model is a simplification of a system built with an intended goal in mind. Models should be easier to use than the original system. This is achieved by abstracting out details of the system that are unnecessary for the target model's intended purpose.

Meta-models: A meta-model is the explicit specification of an abstraction (i.e. model).

Meta-meta-models: A meta-meta-model is used to define meta-models. In particular, the OMG define the MOF (Meta Object Facility) which is a self-defined meta-meta-model.

2.2.2 Model Transformations

There are three main categories of model transformation:

Model-to-model (M2M): M2M transformations are used to translate source models to target models. The source and target models can be instances of the same or different meta-models [18].

Model-to-Text (M2T): These can be considered a special case of M2M transformations. It is often the case that M2T transformations are used for code generation. The most common approach to M2T transformation is a template-based approach. A template contains mixtures of static text and dynamic sections that can be used to access information from the source model [18].

Text-to-Model (T2M): T2M transformations can be used to transform code to a model (conforming to a language meta-model). Model discoverer tools such as MoDisco [11] are the easiest way to perform T2M transformations on code [8].

2.2.3 The Epsilon Framework

Epsilon is a family of programming languages for model management tasks. It provides (among others) the following languages of interest [19]:

Epsilon Object Language (EOL): EOL is the common basis for the languages provided by Epsilon. It can be used for querying and modifying models.

Epsilon Validation Language (EVL): EVL is used for model validation.

Epsilon Transformation Language (ETL): ETL is used for M2M transformations.

Epsilon Generation Language (EGL): EGL is used for M2T transformations.

Epsilon also enables the use of ANT tasks to create workflows of different tasks (e.g. a validation followed by a transformation followed by code generation) [19].
Epsilon has been chosen as the framework for this project over others for the following reasons:

- It is well documented with many tutorials as well as "The Epsilon Book" being freely available. Additionally, there is an active forum for any questions not answered by these sources [20].
- It is well integrated with Eclipse (for example, syntax/error highlighting is provided in the editor and there are graphical tools for running/debugging programs) and has been an official project since 2006 [20].
- Epsilon provides a connectivity layer (EMC). This layer allows EOL programs to access models of different modelling technologies including Eclipse Modelling Framework (EMF) models and XML documents [20].
- I am already familiar with how to use the framework.

2.3 Microcontroller Background

The project involves migration of code from a Parallax Propeller Activity Board to an Arduino Uno. However, since some features of the boards are different, this must be taken into account when migrating code. For example, the Propeller Activity Board uses an 8-core processor whereas the Arduino Uno only has a single core. In this case, the parallelised elements of the source code must be adapted to run on a single core.

The following section highlights the important features and differences between the two boards as well as an overview of the board layout.

2.3.1 Parallax Propeller Activity Board

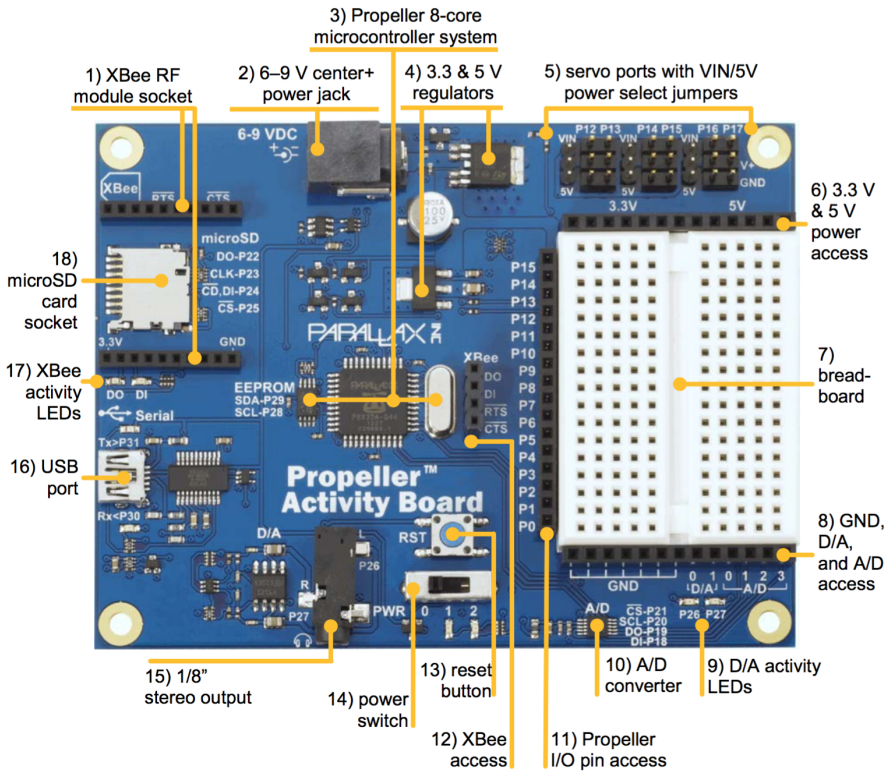

Figure 2.4: Features of the Parallax Propeller Activity Board [21].

The key features of the Parallax Propeller Activity Board are [21]:

- Built-in 8-core Propeller P8X32A microcontroller
- 64KB EEPROM
- XBee wireless module socket
- 16 digital I/O pins
- 4 Analog-to-Digital pins

- 2 Digital-to-Analog pins

2.3.2 Arduino Uno

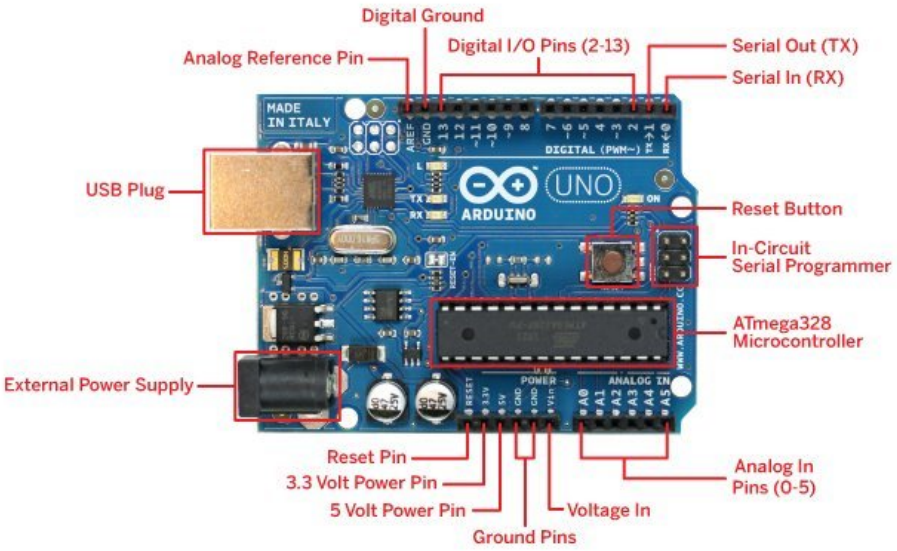

Figure 2.5: Features of the Arduino Uno [22].

The key features of the Arduino Uno are [23]:

- Built-in single-core ATmega328 microcontroller
- 32KB flash memory
- 2KB SRAM
- 1KB EEPROM
- 14 digital I/O pins (6 provide PWM output)
- 6 analog input pins

3 Methodology

3.1 Overview of Methodologies

Project management methodologies can be roughly categorised as either traditional or agile methodologies. Each have their own strengths and weaknesses. Sections 3.1.1 and 3.1.2 provide an overview of these methodologies.

3.1.1 Traditional Methodologies

Traditional approaches include the waterfall/linear methodology and the incremental strategy. The common features of these methodologies are that they are very structured and there is little room for deviation from the original plan. For example, the waterfall methodology splits the project into separate stages for software requirements gathering, systems requirements gathering, analysis, design, coding, testing and operations such that each section is completed before the next begins [24].

These strategies are best suited to projects that have a clear goal and clear solution. In this case, the use of these strategies can have many benefits, including [25]:

- the entire project is scheduled
- resource requirements are known
- team members can be distributed

However, these approaches also exhibit weaknesses, they [25]:

- require heavy documentation
- require a detailed plan
- have difficulty adapting to changes in requirements

3.1.2 Agile Methodologies

Agile methodologies are generally better at accommodating change than traditional methodologies as they repeatedly return to earlier stages in the development process. For example, Figure 3.1 shows a (simplified) view of the stages involved in the iterative, adaptive and extreme agile methodologies. Each approach goes through the full development process, tests and/or deploys the system (or a part of the system) and optionally returns to either the build, design or scope stage depending on the outcome of testing and deployment. This cycle can occur multiple times in the project lifecycle and is referred to as a sprint. Sprints generally last a few weeks but this can vary depending on the project.

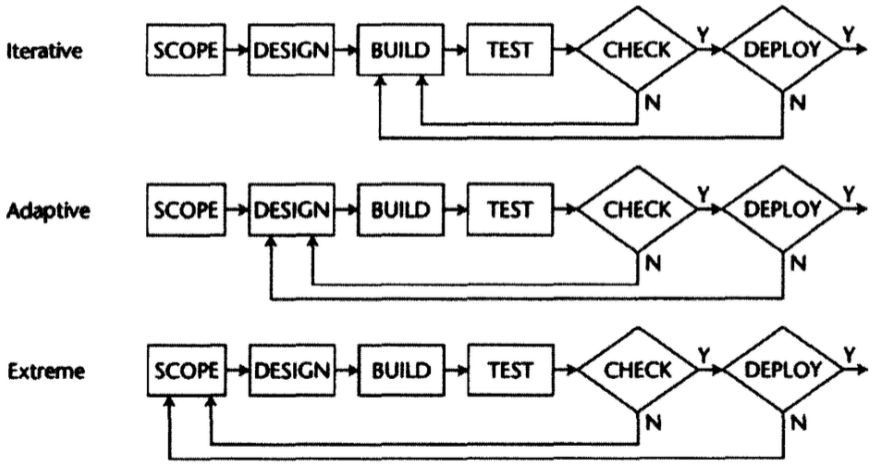

Figure 3.1: An overview of the iterative, adaptive and extreme agile methodologies [25].

As previously mentioned, the ability to return to earlier stages in the development process helps accommodate changing requirements. For example, the client can review what is produced at the end of each sprint and clarify or change requirements depending on the outcome. However, an agile approach requires much closer communication with the client which may not always be possible or desirable. Secondly, since requirements are not fixed for the duration of the project, the final product is not always clear from the start of the project [25].

3.2 Choice of Methodology

The approach for this project is fairly well defined i.e. an MDE-based method will be used to perform software migration. However, the requirements are likely to change throughout the course of the project depending on the success of system prototypes. As such, the project is more suited to an agile methodology as this can more easily manage these changes. Additionally, I will be meeting with the project supervisor weekly which will allow us to work on updating the requirements as necessary and allows the project to be easily split into sprints beginning and ending at these meetings. In particular, the extreme methodology will be the most suitable as it allows for changing the scope which may be required if early research indicates the scope is unsuitable for the time allocated.

3.3 Project Management

This section discusses the tools used in order to manage the project. This includes tools for task management as well as version control tools for the source code and report.

At the start of the project a GANTT chart was created in order to give a rough estimate for when certain parts of the project would be completed. This was not updated throughout the project and doesn't particularly fit with the agile methodology but it was helpful to ensure progress was being made at about the right pace. In particular, this was helpful for writing the report.

The GANTT chart was included in the project page which was created using Google Sites. This page was used for task management and for organising the literature review. Google Sites was chosen as it is free and can be easily adapted to suit the project. The literature review was organised by creating pages on the site for each category within the literature review (e.g. MDE background) and collecting notes from the relevant papers on these pages. The site also contained a list of papers to be read and a list of tasks to be completed.

Git was used as the version control system for both the source code of the system and the documentation as it is free and I already had experience using it. The remote repositories were hosted on GitHub as it allows students to use private repositories for free. In particular, Git was chosen for managing the documentation over e.g. Google Drive as the

report was written in LaTeX and so updates to the main report document would require changes to e.g. image files or the bibliography file. As such, it is easier to backup all these files at once using a single Git command over adding and deleting files from Google Drive. Additionally, the file history is saved using Git so it is easier to return to previous versions of the documentation.

4 Requirements

This chapter discusses the requirements that must be satisfied in order to consider the project successful. Firstly, Section 4.1 introduces the sources used to derive requirements including the relevant stakeholders for the project. Then Sections 4.2 and 4.3 summarise the stakeholder and system requirements respectively. Each requirement is assigned an identifier and a priority based on the MoSCoW categorisation system. In decreasing order, the priority levels each requirement can have are **M** (must have), **S** (should have), **C** (could have) and **W** (won't have).

4.1 Requirements Elicitation

This section discusses the main sources considered for deriving requirements. Firstly, a summary of the initial project statement is presented in Section 4.1.1 as this is helpful for deriving the functional requirements for the project. Secondly, stakeholders for the project are identified in Section 4.1.2 in order to enable gathering of stakeholder requirements.

4.1.1 Project Statement

The initial project statement is as follows:

> "This project aims to design and develop an extensible MDE-based infrastructure for migrating software applications between microprocessors. In particular, the project will enable:
>
> - analysis and extraction of source code deployed on the old microprocessor
> - migration of extracted source code to the new microprocessor"

4.1.2 Stakeholders

Stakeholders are parties that are interested in the development of the project and therefore their goals must be taken into account when deriving

requirements. The rest of this section introduces the roles of the primary stakeholders and gives a high-level overview of their objectives for the project.

Student/Project Developer: Responsible for research, system design, development and testing. Primarily interested in demonstrating the efficacy of an MDE based approach to software migration and successfully completing the project on time.

Project Supervisor: Responsible for meeting regularly with the student and guiding the research for the project. Their goal is also for the system to be finished within schedule.

Software Obsolescence Researchers: Although not directly involved in the project, the results of the project may be of interest to researchers in the field of software obsolescence. In particular, the project explores an alternative approach to the migration of software between microcontrollers presented in [9]. This is a particular problem of interest to the DSTL. However, they are specifically investigating migration between their own microcontrollers such as the FS P4080. Therefore, they would require that the system is easily understandable and can be adapted to work with microcontrollers other than the Parallax Propeller Activity Board and the Arduino Uno.

End Users: No end users are directly involved in the project. However, some people may want to use the resulting system as is to migrate software from a Parallax Propeller Activity Board to an Arduino Uno. In this case, their main priority would be usability of the system. Additionally, it is important that the performance of the migrated software is not significantly worse than the source application. However, as this is a very specific scenario, it is unlikely that this user group will be very large. Similarly to software obsolescence researchers, it may be more useful to end users if the system can be easily modified to support migration between different microcontrollers.

4.2 Stakeholder Requirements

Table 4.1 summarises the stakeholder requirements for the system. Each requirement has an identifier of the form S-X (where X is an integer).

ID	Priority	Description
S-1	M	Users can provide the system with the source code for a Parallax Propeller Activity Board and target code for the Arduino Uno will be generated.
S-2	M	Users can easily adapt the system to support migration of software between different microprocessors.
S-3	S	Users can easily determine sources of errors when using the system.
S-4	S	Users can easily identify areas of the target code that cannot automatically be migrated and must be manually completed.
S-5	S	The performance of the migrated code is not significantly worse than the source application.

Table 4.1: Stakeholder Requirements

4.3 System Requirements

The following section summarises the functional and non-functional requirements of the system. Each functional requirement is assigned an identifier of the form F-X (where X is an integer) and each non-functional requirement is assigned an identifier of the form NF-X (where X is an integer).

Note that valid source code refers to an application which compiles and runs correctly on the source microprocessor and only uses hardware capabilities provided by both microcontrollers.

4.3.1 Functional Requirements

Table 4.2 summarises the functional requirements and their acceptance criteria for the system.

ID	Priority	Description	Acceptance Criteria
F-1	M	The target microcontoller running the migrated code (potentially with manual additions) displays the same behaviour as the source microcontroller, provided the source code is valid.	Manual inspection of the migrated application confirms the behaviour is the same as the source application.
F-2	M	The migrated code deploys without errors on the target microcontroller if the provided source code is valid.	Manual inspection of the migration process confirms a valid source application can be migrated without errors.
F-3	S	The system displays appropriate errors when the source code cannot be migrated (e.g. there are errors in the source code or hardware incompatibilities).	All unit tests pass which check appropriate errors are thrown when invalid source code is provided to the migration process.
F-4	S	The system clearly indicates any errors that have occurred during its use (e.g. the migration process fails).	All unit tests pass which check appropriate errors are thrown when valid source code is provided to the migration process.
F-5	S	The system should indicate any parts of the code that must be manually completed.	Manual inspection of the migrated code confirms that all parts of the code that must be manually completed are clearly indicated (e.g. by TODO directives).

Table 4.2: Functional Requirements

4.3.2 Non-Functional Requirements

Table 4.3 summarises the non-functional requirements and their acceptance criteria for the system.

ID	Priority	Description	Acceptance Criteria
NF-1	M	The system should be easy to adapt to support migration between different microcontrollers.	Manual inspection of the system should indicate that it is easy to adapt. For example, if the system is based on an established MDE framework it will be easier to adapt as there will be more documentation available.
NF-2	S	The system should display a high-level of encapsulation.	Manual inspection of the system should indicate measures have been taken to facilitate encapsulation. For example, separation of hardware-dependent parts of the code.
NF-3	S	The migrated code should not be significantly larger than the source code.	Migrated code is less than 1.5 times the size of the source application.
NF-4	S	The migrated code should exhibit similar performance to the source code.	Migrated code runs in less than 1.5 times the duration of the source application.

Table 4.3: Non-Functional Requirements

4.4 Requirements Traceability Matrix

The requirements traceability matrix shown in Table 4.4 shows how stakeholder requirements correspond to system requirements.

Stakeholder Requirements	System Requirements
S-1	F-1, F-2
S-2	NF-1, NF-2, NF-3
S-3	F-2, F-3, F-4
S-4	F-5
S-5	NF-4

Table 4.4: Requirements Traceability Matrix

5 Design and Implementation

6 Evaluation

7 Conclusion

Bibliography

[1] B. Bartels, U. Ermel, P. Sandborn and M. G. Pecht, *Strategies to the prediction, mitigation and management of product obsolescence*. John Wiley & Sons, 2012, vol. 87.

[2] F. J. R. Rojo, R. Roy and E. Shehab, 'Obsolescence management for long-life contracts: State of the art and future trends', *The international journal of advanced manufacturing technology*, vol. 49, no. 9-12, pp. 1235–1250, 2010.

[3] P. Singh and P. Sandborn, 'Obsolescence driven design refresh planning for sustainment-dominated systems', *The engineering economist*, vol. 51, no. 2, pp. 115–139, 2006.

[4] P. Sandborn, 'Software obsolescence-complicating the part and technology obsolescence management problem', *Ieee transactions on components and packaging technologies*, vol. 30, no. 4, pp. 886–888, 2007.

[5] S. Rajagopal, J. Erkoyuncu and R. Roy, 'Software obsolescence in defence', *Procedia cirp*, vol. 22, pp. 76–80, 2014.

[6] C. Adams, 'Getting a handle on cots obsolescence', *Avionics magazine*, vol. 1, 2005.

[7] P. Sandborn and J. Myers, 'Designing engineering systems for sustainability', in *Handbook of performability engineering*, Springer, 2008, pp. 81–103.

[8] K. Kowalczyk and A. Kwiecinska, *Model-driven software modernization*, 2009.

[9] S. Gerasimou, D. Kolovos, R. Paige and M. Standish, 'Technical obsolescence management strategies for safety-related software for airborne systems', in *Federation of international conferences on software technologies: Applications and foundations*, Springer, 2017, pp. 385–393.

[10] F. Fleurey, E. Breton, B. Baudry, A. Nicolas and J.-M. Jézéquel, 'Model-driven engineering for software migration in a large industrial context', in *International conference on model driven engineering languages and systems*, Springer, 2007, pp. 482–497.

[11] H. Bruneliere, J. Cabot, G. Dupé and F. Madiot, 'Modisco: A model driven reverse engineering framework', *Information and software technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[12] T. Herald, D. Verma, C. Lubert and R. Cloutier, 'An obsolescence management framework for system baseline evolution—perspectives through the system life cycle', *Systems engineering*, vol. 12, no. 1, pp. 1–20, 2009.

[13] J. Wilbrink, 'Facilitating the migration from 8-bit to 32-bit microcontrollers', Atmel Corporation, Tech. Rep., 2004.

[14] D. C. Schmidt, 'Model-driven engineering', *Computer-ieee computer society-*, vol. 39, no. 2, p. 25, 2006.

[15] J. Bézivin, 'In search of a basic principle for model driven engineering', *Novatica journal, special issue*, vol. 5, no. 2, pp. 21–24, 2004.

[16] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill *et al.*, 'The relevance of model-driven engineering thirty years from now', in *International conference on model driven engineering languages and systems*, Springer, 2014, pp. 183–200.

[17] J. Bézivin and O. Gerbé, 'Towards a precise definition of the omg/mda framework', in *Automated software engineering, 2001.(ase 2001). proceedings. 16th annual international conference on*, IEEE, 2001, pp. 273–280.

[18] K. Czarnecki and S. Helsen, 'Classification of model transformation approaches', in *Proceedings of the 2nd oopsla workshop on generative techniques in the context of the model driven architecture*, USA, vol. 45, 2003, pp. 1–17.

[19] D. Kolovos, L. Rose, R. Paige and A. Garcıa-Domınguez, 'The epsilon book', *Structure*, vol. 178, pp. 1–10, 2010.

[20] [Online]. Available: https://www.eclipse.org/epsilon/.

[21] [Online]. Available: https://www.parallax.com/sites/default/files/downloads/32910-Propeller-Activity-Board-Guide-v1.1.pdf.

[22] [Online]. Available: https://0x00sec.org/t/the-hackers-lab-arduino/1708.

[23] [Online]. Available: https://www.farnell.com/datasheets/1682209.pdf.

[24] W. W. Royce, 'Managing the development of large software systems: Concepts and techniques', in *Proceedings of the 9th international conference on software engineering*, IEEE Computer Society Press, 1987, pp. 328–338.

[25] D. J. Fernandez and J. D. Fernandez, 'Agile project management—agilism versus traditional approaches', *Journal of computer information systems*, vol. 49, no. 2, pp. 10–17, 2008.