Supporting Robotic Software Migration Using Static Analysis and Model-Driven Engineering

Sophie Wood
University of York, York, UK
soph.wood52@gmail.com

Nicholas Matragkas
University of York, York, UK
nicholas.matragkas@york.ac.uk

Dimitris Kolovos
University of York, York, UK
dimitris.kolovos@york.ac.uk

Richard Paige
McMaster University, Canada
paigeri@mcmaster.ca

Simos Gerasimou
University of York, York, UK
simos.gerasimou@york.ac.uk

ABSTRACT

The wide use of robotic systems contributed to developing robotic software highly coupled to the hardware platform running the robotic system. Due to increased maintenance cost or changing business priorities, the robotic hardware is infrequently upgraded, thus increasing the risk for technology stagnation. Reducing this risk entails migrating the system and its software to a new hardware platform. Conventional software engineering practices such as complete re-development and code-based migration, albeit useful in mitigating these obsolescence issues, they are time-consuming and overly expensive. Our RoboSMi model-driven approach supports the migration of the software controlling a robotic system between hardware platforms. First, RoboSMi executes static analysis on the robotic software of the source hardware platform to identify platform-dependent and platform-agnostic software constructs. By analysing a model that expresses the architecture of robotic components on the target platform, RoboSMi establishes the hardware configuration of those components and suggests software libraries for each component whose execution will enable the robotic software to control the components. Finally, RoboSMi through code-generation produces software for the target platform and indicates areas that require manual intervention by robotic engineers to complete the migration. We evaluate the applicability of RoboSMi and analyse the level of automation and performance provided from its use by migrating two robotic systems deployed for an environmental monitoring and a line following mission from a Propeller Activity Board to an Arduino Uno.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; Robotics; • **Software and its engineering → Software evolution**.

KEYWORDS

model-driven engineering, robotic systems, software migration, static analysis

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410965

ACM Reference Format:
Sophie Wood, Nicholas Matragkas, Dimitris Kolovos, Richard Paige, and Simos Gerasimou. 2020. Supporting Robotic Software Migration Using Static Analysis and Model-Driven Engineering. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410965

1 INTRODUCTION

Robotic systems are increasingly used in various application domains ranging from transportation [11] and healthcare [17] to agriculture [5] and warehouse management [34]. Driven by recent technological advancements, these systems provide sophisticated functionality, increased efficiency and automation by assisting or, when possible, replacing human operators in repetitive, laborious or potentially dangerous tasks [32]. For instance, mobile robots deployed within a warehouse facility can perform automatic inventory checks, thus reducing the need for manual inventory counts. These robots enable cost-effective inventory tracking by supporting inventory identification in put-away locations, near real-time analysis and visualisation of product storage. Recent reports highlight the significant social and economic benefits of using mobile robotics in industrial environments and the public domain [18, 24].

Despite the potential robotics-induced benefits, more often than not, the robotic systems are underpinned by fragile designs leading to unavoidable obsolescence issues [16, 20]. The reasons for this fragility are primarily *technological*, resulting in an architectural gap between system components [3]. The continuous development of new hardware and software components that provide capability and performance improvements reduces the long-term support of legacy components. Although the generic architecture of modern robotic systems enables hardware components to be put together in a 'plug and play' manner, hence, motivating system upgrade with more technologically advanced components, the significant cost incurred is an inhibiting factor [31]. Other reasons contributing to fragile designs include changes in functional or non-functional requirements (such as new timing requirements) leading to incompatibility between system components.

Selecting an appropriate modernisation strategy is undoubtedly critical for reducing the risk of technology stagnation and modernising the robotic system successfully [1, 2, 27]. Pursuing a complete re-development strategy, albeit useful since it drives re-designing, refactoring and customisation of the legacy robotic system, incurs unrealistic costs that are directly proportional to the

size of the system [12]. Similarly, adopting a conventional code-based migration approach is not only time-consuming but also error-prone since it ignores rich models (e.g., hardware architecture diagrams) that are typically produced by the design team of an organisation during the early stages of system development. In contrast, employing a *model-based strategy* that facilitates the migration of the robotic system to a new hardware platform using available architectural models offers several benefits, including better maintainability, reduced cost and reusability [26].

In this paper, we introduce RoboSMi, a model-driven software migration approach that supports migration of the software controlling a robotic system between hardware platforms. First, RoboSMi uses static analysis on the robotic software of the source hardware platform to extract platform-specific and platform-agnostic constructs. The static analysis is driven by the C/C++ Development Tooling (CDT) model driver of the Epsilon platform [23] specifically developed to support RoboSMi. Through transformations of models capturing the hardware architecture of components on the target platform (e.g., temperature sensors, servo motors), RoboSMi determines key attributes for those components (e.g., sensor type, interfaces used to communicate with the target platform). For each component, RoboSMi produces a ranked list of candidate software libraries suitable for the target platform, by employing the TFIDF statistical measure for information retrieval [30] and combining extracted information for this component and a platform-specific repository enhanced with historical data. Finally, RoboSMi uses code generation to refactor the software, making it suitable for the target platform by realising the adapter pattern [13], highlight platform-specific constructs that need manual edits and add software constructs to support the invocation of hardware components on the target platform. Engineers can complete the migration by inspecting the highlighted areas and adding source code that enables the invocation of hardware components on the target platform.

We evaluate RoboSMi by migrating two robotic systems deployed for a line following and an environmental monitoring mission from a Propeller Activity Board to an Arduino Uno microcontroller [35]. Our experimental evaluation shows that RoboSMi can support the modernisation of robotic systems by making suitable modifications to the software running on the source platform. The manual adaptations needed to complete the migration and develop a fully-fledged robotic system for Arduino focus on inferring constructs (i.e., invocations to Arduino libraries and auxiliary code) that achieve the same functionality as the source system. We did not observe any perceptible performance difference when deploying the robotic system on the target platform.

The main contributions of our paper are as follows:

- The RoboSMi approach enabling the migration of robotics software between hardware platforms (Section 3);
- The Epsilon Model Connectivity (EMC) CDT model driver that enables the management of C/C++ software using MDE technologies specifically developed to support RoboSMi;
- The open-source RoboSMi prototype tool presented in Section 4 and made available on our project webpage;
- The evaluation of the RoboSMi approach and prototype tool summarised in Section 5.

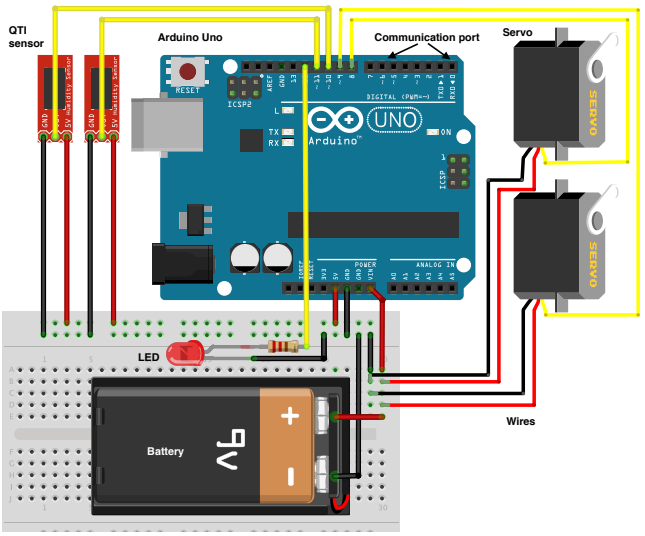

Figure 1: Architecture of the line following robotic application in Fritzing.

Listing 1: Software excerpt of the line following robotic application running on the Propeller Activity Bot.

```
#include "simpletools.h"
#include "abdrive.h"
const int leftQTI  = 6;  //Left QTI port
const int rightQTI = 7;  //Right QTI port
const int LED      = 8;  //LED port
const int QTI_T    = 50; //QTI sensor threshold
const int waitTime = 230;//QTI wait time
long GetQTIState(int comPort){
 set_output(comPort, 0b1);
 set_direction(comPort, 0b1);
 waitcnt(waitTime);
 set_direction(comPort, 0b0);
 waitcnt(waitTime);
 long state = get_state(comPort);
 return state;
}
int main(){
 while (1){
  long sLeft  = GetQTIState(leftQTI);
  long sRight = GetQTIState(rightQTI);
  if (sLeft < QTI_T && sRight < QTI_T)
   drive_speed(24, 24); //continue straight
  else if(sLeft < QTI_T && sRight > QTI_T)
   drive_speed(24, 12); //turn left
  else if(sLeft > QTI_T && sRight > QTI_T){
   drive_speed(0, 0);  //stop
   high(LED);          //switch on LED
   pause(1000);         //wait 1s
   low(LED);           //switch off LED
   drive_speed(24, 24);//continue
  }
}
...
```

2 MOTIVATING EXAMPLE

We will illustrate the RoboSMi approach for the model-driven migration of robotics software using a robotic application deployed to navigate across the floor of a building by following a dark line. The robotic application has the target architecture shown in Figure 1 developed using the open-source electronic computer-aided design (ECAD) tool Fritzing [21]. Similar to other ECAD tools (e.g., KiCad, Eagle), Fritzing enables to design schematics of electronic-based prototypes by selecting hardware components (e.g., an Arduino Uno hardware platform, temperature sensors, servos, resistors) and

wiring these components together through their communication ports to generate a closed circuit that performs a specific task. The generated schematics can be used to produce printed circuit boards for hardware-in-the-loop simulation or for manufacturing and commercialisation purposes.

The robotic application uses an Arduino Uno as its target hardware platform for controlling the execution of the application, moves using two servos and maintains its position on the dark line using two QTI sensors. These sensors can identify light and dark surfaces (e.g., a black line drawn on a white floor) through using an infrared light-emitting diode (LED) and an infrared phototransistor internally. The light emitted by the LED bounces off a surface and is detected by the phototransistor. Surfaces with darker colour absorb the emitted light, and thus the phototransistor gives lower values. Once a checkpoint is discovered (i.e., both QTI sensors detect a dark mark), the robot emits light through the red LED. The resistor between the LED and Arduino is a passive component that protects the LED by limiting the flow of electrical current when a voltage is applied across it. Finally, a 9V battery provides power to the robotic application.

Listing 1 shows an excerpt of the C software for this robotic application deployed on a Propeller Activity Bot. The initial segment of this software includes the platform-specific libraries through the include directives (lines 1-2) and the declaration of the communication ports for the robot components (lines 3-5). The `GetQTIState` function emits infrared light (lines 9-10) and calculates the amount of light detected by a phototransistor when reflected off by a nearby surface by measuring the rate of charge transfer through the phototransistor (lines 11-15). While running, the application executes the `while` loop whose purpose is to determine the values of the QTI sensors (by invoking the `GetQTIState` function) and, depending on whether these values are below or above an engineer-defined QTI threshold (line 6), to steer the robot accordingly (e.g., straight, left, right) by invoking the `drive_speed` function of the servos (lines 21-31). When the robot encounters a checkpoint, it stops and switches on the red LED for one second, before switching off the LED and continuing its journey.

Although this robotic application might seemingly look simple, the manual migration of the software from the Propeller Activity Board to a more powerful hardware platform (e.g., an Arduino Uno) is a challenging activity. In particular, this activity involves analysing the original software to determine which code fragments must be migrated and which must be deleted, identifying platform-specific libraries and modifying the communication ports to match the new platform. In the following section, we present the RoboSMi approach and illustrate its ability to automate important steps of the migration activity.

3 APPROACH

In this section, we describe our RoboSMi model-driven approach for the systematic migration of software between robotic platforms. RoboSMi, whose high-level workflow is shown in Figure 2, uses the following artefacts: (i) the software deployed on the source robotic platform that will undergo the migration process (green solid box); (ii) a model specifying the hardware architecture for the target robotic platform (blue dashed box); and (iii) a platform-specific repository that holds key information about the target

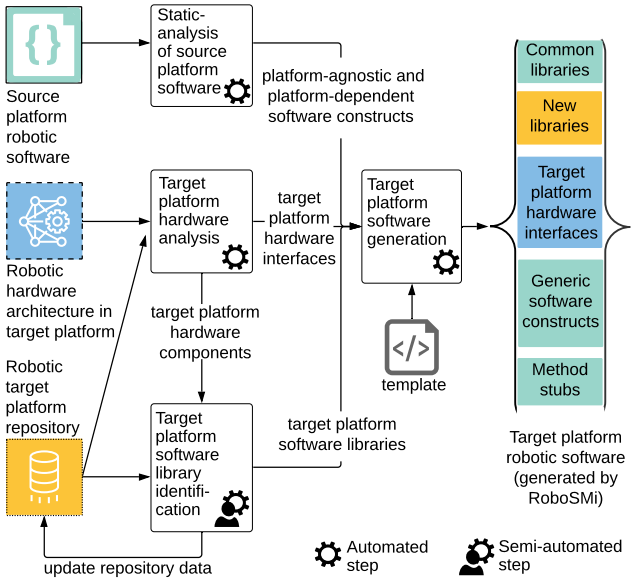

Figure 2: RoboSMi workflow.

platform (yellow dotted box). We assume that no model of the source platform hardware is available. This is a common issue for most modernisation activities as these legacy hardware specifications are often incompatible with currently used tools, might require tacit knowledge or have been misplaced.

RoboSMi starts with analysing the software deployed on the source robotic platform to determine generic (i.e., platform-agnostic) and platform-dependent software constructs. These constructs form the input for the model-to-text transformation of software for the target platform. Next, through examining a hardware specification of the target robotic platform, RoboSMi identifies information about the target hardware platform that is essential for the transformation. This set of information includes the types and characteristics of hardware components and interfaces used for communicating with the target platform. The next step of the approach involves combining the extracted information with information from the platform-specific repository to recommend suitable software libraries whose selection will enable the modernised code to access those hardware components. Finally, using all extracted key data from the previous three steps, the software for the target platform is generated through model-to-text transformation and realisation of the adapter design pattern [13]. Although the generated code is fully compilable (i.e., an executable can be generated), software engineers should inspect it and complete the empty placeholders with suitable code (cf. Section 3.4). In the following sections, we describe the general principles underpinning each step of RoboSMi and present its realisation for the tool-supported RoboSMi instance.

3.1 Source Platform Software Static-Analysis

Approach. During this automated step, RoboSMi analyses the software of the source platform to establish the set of modifications needed to make the software suitable for the target platform. To this end, the source software undergoes a series of static analysis actions including scanning and preprocessing, semantic analysis, name resolution and binding, that enables to generate its Abstract

Syntax Trees (ASTs). Once parsing finishes, the generated ASTs are analysed using AST visitor implementations, to detect platform-agnostic and platform-dependent software constructs. Platform-agnostic constructs will remain unchanged during the software generation for the target platform. For instance, include directives that enable accessing standard C/C++ library functions such as containers (e.g., queue, set) and input/output streams (e.g., stdio, fstream) are common across both platforms and will not be modified. In contrast, platform-dependent constructs (e.g., methods native to the source platform) require manual adaptations. Consequently, these constructs are marked and will be handled accordingly in the model-to-text transformation step (cf. Section 3.4).

Realisation. The tool-supported RoboSMi instance carries out the static analysis step using a combination of lightweight model querying and transformation operations. More specifically, RoboSMi uses the software as a model and navigates through it on-demand [19]. Thus, the entire analysis and extraction of software constructs is guided through a set of model management queries at the abstract syntax level defined in the Epsilon Object Language (EOL) [22]. This alleviates the need to transform the source platform software into an EMF-compatible representation, a task that not only requires accurate model extraction tools and a complete metamodel of the analysed general-purpose programming language, but it is also time-consuming for large software systems [6].

The RoboSMi analysis is underpinned by the EMC CDT model driver [15], a "technology-specific driver" that exposes the document object model (DOM) maintained by the Eclipse C/C++ Developer Tools (CDT) [9] as models. Accordingly, the EMC CDT driver provides access to the internal representations maintained by CDT in the form of models that are suitable for the model management languages of the Epsilon platform [22]. Once a query is made (e.g., to identify platform-independent include directives), the EMC CDT driver employs a ReflectiveASTVisitor, a specialisation of the visitor design pattern [13], to parse the relevant software and perform the necessary binding resolution, to traverse the DOM abstract syntax tree and, finally, to return all software constructs meeting the constraint (e.g., include directives). A similar process is applied for queries that extract platform-dependent constructs. RoboSMi caches the results in memory to reduce parsing and analysis times of similar queries in the future.

Example 3.1. Consider Listing 1 of the robotic application (Section 2). The functions `set_output`, `set_direction`, `waitcnt`, `high`, `low`, `get_state`, `drive_speed`, and `pause` are platform-dependent and are identified by RoboSMi as constructs for modernisation. The include directives (lines 1-2) are specific to the Propeller platform and are not marked for the model-to-text generation step (cf. Section 3.4). The remaining constructs (e.g., variable declarations on lines 5-10, function declarations on lines 12, 25 and 30) are platform-independent and will not be modified during the modernisation.

3.2 Target Platform Hardware Analysis

Approach. In this RoboSMi step, a diagrammatic specification of the system architecture for the target hardware platform is used for the extraction of key information of the components comprising the target platform. This is an important step as each platform has its own architecture with specific communication interfaces (e.g., ports) for hardware components (e.g., robotic servos), power access points, and software libraries for digital and analog input/output. Hence, the migration of the robotic application requires to correctly determine the type of used hardware components (i.e., models of sensors and actuators) and to establish the communication interfaces through which the software can interact with the components on the target platform. Since the objective is the generation of a functionally-equivalent application for the target platform, identifying the hardware components must be enhanced with information of communication interfaces with the target platform. The manual execution of these tasks is challenging and error-prone, especially when different teams undertake the generation of the diagrammatic specification and software development. This is a common scenario in large organisations that causes the software team to spend a considerable amount of time to study the diagrammatic specification before proceeding with the migration. RoboSMi automates this step by transforming the diagrammatic specification of the target platform configuration into an EMF compatible model that is suitable for model management operations. Given the corresponding EMF model, RoboSMi executes model queries to extract both the precise model of employed hardware components (used in the next RoboSMi step for software library recommendation) and the communication interfaces from the target hardware platform.

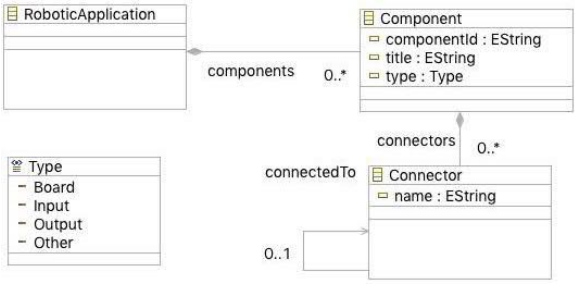

Figure 3: Metamodel of a robotic application

Realisation. The tool-supported RoboSMi instance uses as input a Fritzing specification [21] of the robotic application for the target hardware platform. Figure 1 shows the Fritzing specification of the line following robotic application introduced in Section 2. Given a Fritzing project as an XML file, RoboSMi carries out a text-to-model transformation to instantiate the metamodel shown in Figure 3. This robot-specific metamodel represents the configuration of a robotic application as a set of components. Each component is assigned a specific type (e.g., Board, Input, Output) and has a set of connectors through which it can connect to (i.e., communicate) other components on the platform. The type of each component is extracted from a platform-specific repository which records whether the component provides input (e.g., a temperature sensor), output (e.g., a servo) or auxiliary (e.g., a resistor) functionality. The main component is the hardware platform that runs the robotic application and is of type Board. The connectors of this component include all the digital and analog interfaces through which the platform can communicate with other components. Extracting this information by directly analysing the XML file (e.g., through the XML driver for Epsilon [23]), albeit possible, it introduces additional complexity, requires redundant analyses of the XML file, and would make the

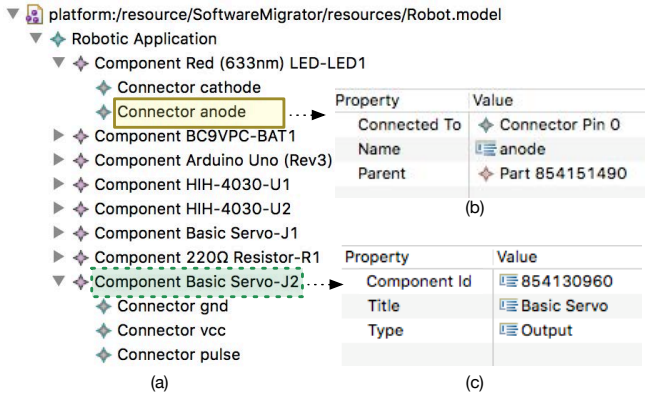

Figure 4: Model instance (a) of the line following robotic application from Section 2 corresponding to the metamodel from Figure 3 with information about the LED anode connector (b) and the servo component (c).

combination of information from the platform-specific repository convoluted. Figure 4 shows the EMF model for the line following robotic application (Figure 1) along with extracted information for the hardware components and connectors.

Having generated the EMF model instance of the Fritzing specification, RoboSMi analyses thoroughly the model to determine a mapping set that signifies how connectors from the hardware platform are connected with components that perform input and output functionality (i.e., sensors and actuators). Setting the connectors correctly is important both for having a valid hardware specification and for enabling the software to access and control the hardware components. Incorrect connector specification could damage the hardware components or the entire hardware platform.

RoboSMi executes Algorithm 1 to generate the mappings. Given the connectors on the hardware platform by the GetConnectors function (line 3), RoboSMi selects each connector c in sequence and analyses the transitive association with other connectors until it reaches a connector whose parent is an (active) input or output component (lines 10–11). To establish this transitive association, RoboSMi applies a depth first search strategy as only connectors leading to input or output components are really accessible through the software. In contrast, passive components (e.g., resistors) should be traversed to determine the input/output components with which they are connected. For instance, resistors are necessary for limiting the flow of electrical current in an electronic circuit or for providing a specific voltage to another component (e.g., temperature sensors, servos). Despite their importance, the value of these components in the software side is on identifying the input/output components they "protect" through analysing their connectors. Thus, when these passive components are encountered during model traversal, RoboSMi adds their connectors to the set of connectors for analysis (lines 14– 15). Algorithm 1 can also establish the mappings with components requiring multiple connections to the hardware platform (e.g., a liquid crystal display requires six connections). Once the algorithm completes, it returns the MAPPINGS data structure that denotes how connectors from the hardware platform are connected to connectors from input/output components (line 16).

Complexity analysis. We can define the system architecture as a graph with N nodes and E edges. The nodes $N^C \subseteq N$ belong to

Algorithm 1 Platform component identification

```
1:  function AnalyseHardwarePlatform(P)
2:      MAPPINGS ← ∅
3:      for all c ∈ GetConnectors(P) do
4:          S ← {c}
5:          V ← ∅
6:          while ¬(S = ∅) do
7:              d ← SelectFirst(S)
8:              if d ∉ V then
9:                  V = V ∪ {d}
10:                 if Type(Parent(d)) ∈ {Input, Output} then
11:                     MAPPINGS ← Append(c, d)
12:                 else
13:                     S ← S ∪ {ConnectedTo(d)}
14:                     if Type(Parent(d)) ∉ {Board} then
15:                         S ← S ∪ GetConnectors(Parent(d))
16:     return MAPPINGS
```

the hardware platform P. For each $c \in N^C$, Algorithm 1 executes a depth-first search until finding an edge whose ending node is an input or output component. Consequently, the worst-case performance for graph traversal without repetition is $N^C \times O(|N| + |E|)$.

Example 3.2. Consider the Fritzing specification of the line following robotic application from Figure 1. The execution of this RoboSMi step generates the model in Figure 4. The text next to a component corresponds to its Fritzing-specific unique identifier. Algorithm 1 produces the MAPPINGS structure {BasicServo2WriteP1→8, BasicServo1WriteP1→9, HIH40302ReadP1→10, HIH40301ReadP1 →11, Red633nm-LED1WriteP1→12} that maps a unique identifier for each connector of every component to the hardware platform connector it is connected. For instance, {Red633nm-LED1WriteP1→12} denotes that the anode connector of the red LED (Figure. 4a) is connected to connector #12 of Arduino Uno. This mapping has been established through traversing the resistor's connectors between the LED and Arduino.

3.3 Target Platform Software Library Identification

Approach. In this RoboSMi step, candidate software libraries for every hardware component of the target platform are identified and recommended to engineers. Since hardware platforms are typically packaged with several software libraries[1] that enable interacting with components deployed on the platform, selecting the most suitable library for each component by manually inspecting those libraries is a non-trivial task.

RoboSMi facilitates software library selection by recommending a set of candidate software libraries for the target platform for each hardware component by exploiting the information available about the software libraries and the components. More specifically, RoboSMi combines key information for each component identified through model analysis of the Fritzing specification (Section 3.2) with information about software libraries for the target platform and historical data of previously used libraries for the component, both available in the target platform repository. After the most likely software libraries for each component have been identified,

[1] e.g., https://www.arduino.cc/enReference/Libraries

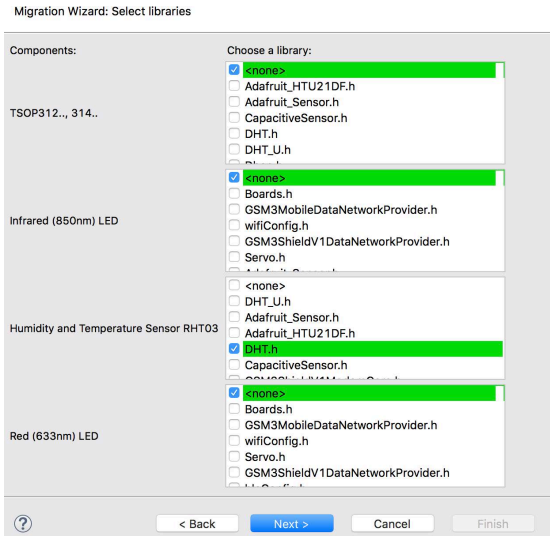

Figure 5: Ranked software libraries per component for the line following robotic application from Section 2.

they are presented to engineers through an intuitive user interface (Figure 5) for selecting the library to realise the functionality of the component. Once the selection task is completed, information about the selected libraries is forwarded to the final RoboSMi step for the generation of the target platform software.

Realisation. The tool-supported RoboSMi instance produces a ranked list of candidate software libraries for each hardware component using a two-phase method. During the first phase, RoboSMi transforms the ranking into an information retrieval task that aims at establishing the relevance score of a library based on a component's name and a collection of header files corresponding to software libraries available for the target platform. A header file holds software constructs of a library (e.g., function and variable declarations) and represents its publicly available interface via which third-party client software (e.g., the software for the target platform) can invoke the functionality of the library. Given a hardware component, RoboSMi establishes for each library its term frequency-inverse document frequency (*TFIDF*) score [30] given by

$$\text{TDIDF}_d = \sum_{t \in T} \left(1 + log(tf_{f,d})\right) \times log\left(\frac{N}{df_t}\right) \quad (1)$$

where T : the set of terms
df_t : the number of documents term t occurs in
N : the number of documents in the collection
$tf_{t,d}$: occurrences of term t in document d

TFIDF weighs a term's frequency (TF) and its inverse document frequency (IDF) with each term having its respective TF and IDF score. In RoboSMi, a document corresponds to a header file and a term corresponds to a part in the name of a component. The total ranking for a library is calculated by summing the TFIDF score for each term in the name. TFIDF takes into account both the frequency of occurrence of a term in the library as well as its "informativeness". As such, a term is more informative if it appears in fewer documents of the collection; "informativeness" effectively corresponds to the $log\left(\frac{N}{df_t}\right)$ term in (1). For example, if we consider the QTI sensor in the line following robotic application, the term *sensor* is less informative than the term *QTI* as many components have the term *sensor* in their name and so will many header files. If "informativeness" is not taken into account, any component that contains the term *sensor* in its name is likely to rank higher than the QTI sensor component, since the term *sensor* will have the most occurrences in the text, regardless of the type of sensor. Since *QTI* is a less common term, using the TFIDF score will instead rank header files containing the less frequently occurring term *QTI* higher.

The accuracy of rankings depends on the most suitable libraries for a component making some reference or including similar terms to the component's name. Given the information provided in the Fritzing specification (cf. Section 3.2), the name of a component is the only information that could be used for determining the TFIDF score. Exploring other statistical measure techniques and improving the accuracy of TDIDF is part of our future work (cf. Section 7).

To improve further the quality of library recommendations, RoboSMi in its second phase executes a set of queries to a platform-specific database of historical configuration data to establish whether the component has been used previously in another migration activity. If this holds, RoboSMi extracts the chosen libraries for this component along with a counter indicating how many times each library has been chosen. The higher the number, the more likely this library will be selected for this migration activity.

By using a utility function that combines the TDIDF scores and the information from the historical configuration data, RoboSMi generates a ranked list of candidate libraries for each hardware component. Engineers can review this list and select the libraries matching the system's components (Figure 5). Once a selection has been made, the configuration data is updated accordingly. Since the correct library is expected to be chosen most frequently, over time this historical configuration data will improve the accuracy of library suggestions.

3.4 Target Platform Software Generation

Approach. The last RoboSMi step involves the automatic generation of software for the target platform by instantiating a software template using information extracted from the previous three steps (see Figure 2). In particular, for each source file provided as input from the source platform, RoboSMi generates the corresponding source file for the target platform in which platform-agnostic software constructs are retained (Section 3.1), information about the ports for interfacing with the hardware components is added (Section 3.2) as well as include directives for the selected software libraries (Section 3.3). For platform-specific constructs, extracted from analysing the source platform software (Section 3.1), RoboSMi implements the adapter design pattern [13] and generates placeholders using the signatures of these constructs enhanced with suitable TODO directives. Using this pattern reduces the changes to the source software while delegating the modifications to start exercising the functionality on the target platform to the produced placeholders [4, 33]. This pattern has also been applied for the integration of legacy systems using MDE [7]. Having a fully working software for the target platform requires to populate the empty placeholders with suitable code (e.g., invocations to methods of selected libraries). The TODO directives, summarised into a task

Listing 2: EGL template except for generating the target platform software.

```
1 [*Add platform-agnostic libs*]
2 [% for (i in %getPlatformAgnosticLibraries()%){%]
3 #include ``[%=i%]''[%}%]
4 [*Init port interfaces*]
5 [% var ports = %generatePortInterfaces()%;%]
6 [% for (p in ports.keySet()) {%]
7 const int [%=p%] = [%=ports.get(p)%]; [%}%]
8 [*Add platform-dependent constructs*]
9 [% for (d in %getPlatformDependConstructs()%){%]
10 [%= d.getDeclarator().getRawSignature() %]
11 {\n//TODO: complete method\n}
12 [%}%]
```

Listing 3: Generated software excerpt of the line following robotic application (the colouring scheme corresponds to the semantics in Figure 2)

```
1 #include <stdio.h>
2 #include "Arduino.h"
3 #include <Servo.h>
4 const int QTI_T    = 100; //QTI sensor threshold
5 const int leftQTI    = 10; //Left QTI port
6 const int rightQTI   = 11; //Right QTI port
7 const int leftSrvPrt  = 8;  //Left servo port
8 const int rightSrvPrt = 9;  //Right servo port
9 const int LED         = 12; //LED port
10 long GetQTIState(int comPort){
11  ...
12 }
13 void setup() {//Arduino specific(startup)
14  ...
15 }
16 void loop() {//Arduino specific(running forever)
17  int sLeft  = GetQTIState(leftQTI);
18  int sRight = GetQTIState(rightQTI);
19  if (sLeft < QTI_T && sRight < QTI_T)
20   drive_speed(24, 24); //continue straight
21  ...
22 }
23 void drive_speed(int left, int right){
24  //TODO: complete method
25 }
26 ...
```

list, provide software engineers with a clear view of parts of the generated software that require manual inspection, refactoring or completion to finalise the migration to the target platform.

Realisation. The tool-supported RoboSMi instance generates the software for the target platform by executing a model-to-text transformation in the Epsilon Generation (EGL) Language [29]. Listing 2 shows an excerpt of the transformation that generates the include directives for the platform-agnostic libraries (lines 1-3), the constructs for communicating with the hardware components (lines 4-7) and the implementation of the adapter pattern for the platform-specific methods found (lines 8-12).

Example 3.3. Consider again the Propeller Activity Board software shown in Listing 1. The model-to-text transformation generates the software excerpt shown in Listing 3. The colouring scheme matches the colours from the RoboSMi workflow (Figure 2) and signifies the output of the RoboSMi step from which the corresponding code has been derived. For instance, the include directives (lines 2-3; shown in blue) correspond to the libraries selected in Figure 5 and the `drive_speed` method definition (lines 28-30) represents the platform-specific construct derived from source platform software analysis (Section 3.1). The `setup` and `loop` methods are required by the Arduino Uno platform and are included in the EGL transformation (omitted from Listing 2 due to space constraints).

4 ROBOSMI PROTOTYPE ECLIPSE PLUGIN

To automate the software migration process, we implemented a prototype tool as an Eclipse plugin with the architecture shown in Figure 2. Our RoboSMi tool uses the Epsilon framework [23] in all its components and for all model management tasks including model querying and transformations. Finally, the static analysis component uses the Eclipse CDT Project [9] and Epsilon CDT driver [15] for parsing the abstract syntax tree of the source platform software and using C/C++ models with Epsilon. The open-source RoboSMi code, the full experimental results summarised in the following section, additional information about RoboSMi and the case studies used for its evaluation are available as open-source at https://github.com/gerasimou/RoboSMi.

5 EVALUATION

5.1 Research Questions

RQ1 (Validation): Can RoboSMi support migration between hardware platforms? We used this research question to establish if RoboSMi can generate the required artefacts and support the modernisation of robotic systems.

RQ2 (Automation Level): What is the level of automation supported by RoboSMi for this migration activity? Since RoboSMi aims to reduce the effort required for software migration in robotic systems, we examined the manual code that remains to be written to complete the migration task.

RQ3 (Performance): What is the impact of the adapter pattern in software migration? We analyse the extent to which the placeholders generated due to the adapter pattern affects the completion of the migration to the target platform.

RQ4 (Generality): What is the generality level of RoboSMi? We elaborate on the extent to which the RoboSMi approach can support software migration tasks in similar application domains.

5.2 Experimental Setup

We evaluate the end-to-end RoboSMi behaviour using two robotic applications similar to those available on open-source robotic repositories (e.g., https://create.arduino.cc/projecthub). The first is the line following application from Section 2. The second involves a robot deployed for an environmental monitoring mission. In particular, the robot, shown in Figure 6, can navigate its surroundings using its servo motors until a system of infrared transmitters and receivers detects objects blocking its path. When an obstacle is detected, the robot stops, rotates and continues moving in another direction. Also, the robot takes periodic humidity and temperature readings of its operating environment using a DHT22 sensor. When any of these environmental attributes is above a predefined threshold, an LED is triggered to alert any interested stakeholders. The Fritzing specification is available on the project webpage.

Both applications are originally deployed on robots available in our lab using a Propeller Activity board and are migrated to

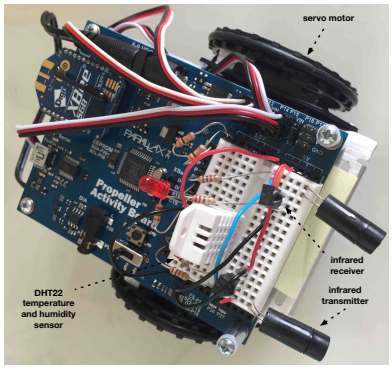

Figure 6: Environmental monitoring robotic application running on the Propeller Activity board (https://www.parallax.com/product/32912).

functionally-equivalent robots using an Arduino Uno. As such, the source platform software contains a mixture of Propeller library functions, as well as global variables, user-defined functions, and include statements for non-Propeller libraries. The hardware configuration in both platforms has the same hardware components but with some variations in the communication ports with which components connect to platforms. This is due to platform-specific constraints, e.g., port #12 on the Propeller is reserved for the left servo encoder whereas this port can support any component on an Arduino.

5.3 Results and Discussion

RQ1 (Validation). We carried out the experiments described in the previous section and confirmed that RoboSMi produced the artifacts for all its steps as presented in Sections 3.1–3.4. More specifically, the source platform software analysis step partitioned the software for both applications into Propeller-specific commands (e.g., include directives, method invocations to Propeller software libraries) and platform-agnostic commands (e.g., global variable declarations, locally defined and used variables and methods).

By inspecting the Fritzing specification for each application, RoboSMi generated the EMF model instances with characteristics of the hardware components (cf. Figure 4) and established the mappings set of connections between Arduino Uno (target platform) and the components. An interesting challenge we dealt with in this step was the extraction of a component's type (e.g., Input, Output) due to ambiguous and inconsistent information in the Fritzing specification. For instance, the distinguishing attribute in the Fritzing specification for the QTI sensor, DHT22 temperature and humidity sensor, infrared receiver, and servo motor had the values "out", "data-signal", "data", and "pulse", respectively. Consequently, establishing the functionality type provided by a component using only this information is not trivial. We mitigated this challenge by extracting information from an Arduino-specific repository, incrementally developed during the course of the experiments.

Formulating the identification of mappings set in this second RoboSMi step as a graph traversal problem and solving it using Algorithm 1 enables to address several challenges. First, by considering passive hardware components (e.g., resistors, potentiometers) as branch points (i.e., with at least one possible outgoing path) and applying a depth-first strategy, we establish the transitive association between ports on input/output components and the platform (cf. Example 3.2). Furthermore, the graph traversal problem formulation facilitates the identification of ports for components that require multiple ports on the platform. For example, an LCD needs six ports from the platform (four for data, and two for enabling and activating the component). The depth-first strategy is also particularly useful when ports on the target platform are common among multiple components. This scenario can occur when the components are chained together and controlled by the same port. For the environmental monitoring application, for example, the port controlling the LED could be also used by a piezoelectric speaker that starts emitting sound upon exceeding the predefined threshold. Components are also chained together when employing a communication protocol other than the general-purpose input/output (GPIO), e.g., serial peripheral interface (SPI), inter-integrated circuit (I2C). For instance, the I2C protocol uses a master-slave setup and requires only two ports to connect up to 1023 components (slaves) onto the same platform (master). Thus, RoboSMi Algorithm 1 also automates the identification of ports for components using different communication protocols (GPIO, SPI, and I2C).

The list of candidate software libraries produced in the third RoboSMi step (cf. Section 3.3) ranked the correct libraries among the top five (out of 247 in total) per application. Thus, the transformation of library identification into an information retrieval task and the calculation of TFIDF-based (1) relevance scores provided practicable recommendations. For instance, the correct libraries `Servo.h` and `DHT.h` for the servo motor and the DHT22 sensor in the environmental monitoring application are ranked second and fourth, respectively. For DHT22, the more highly suggested libraries are all for humidity/temperature sensors, although not for this specific sensor.

Since TDIDF combines information from the name of a component and the contents of the library's header, its relevance score presents interesting sensitivity attributes. In particular, the name given to a component by Fritzing is particularly important. For example, the infrared receiver is named as "TSOP312.." which corresponds to a vendor-specific component and gives little indication on the functionality provided by the component. Consequently, the recommendations for this component are not useful. Another example concerns the servo motor for which the `Boards.h` library is ranked higher than `Servo.h`. This occurs because `Boards.h` contains macro definitions (e.g., `MAX_SERVOS`) for different types of boards (e.g. Arduino Mega, Uno). Hence the term "servo" appears many times, contributing to a higher TFIDF score. The use of historical configuration data partially addresses these issues. If the correct library has been chosen in the past, it will be ranked higher in subsequent RoboSMi use.

The instantiation of the adapter pattern to generate the Arduino Uno software in the final RoboSMi step (cf. Section 3.4) produces the expected output. To this end, suitable include directives are defined for platform-agnostic libraries and those selected in the previous step (cf. Section 3.3), and placeholder methods with TODO directives are generated for any invoked Propeller-specific library method. Also, commands from the main method outside the `while(1)` loop are copied to the `setup` method, commands within `while(1)` are copied to the `loop` method, any other

defined methods and global variables are copied to the generated software.

RQ2 (Automation Level). To answer this research question, we completed the migration and developed fully-fledged robotic applications for the Arduino platform by manually adapting the RoboSMi outputs and populating the placeholders with suitable commands. We confirmed the functional equivalence of the migrated applications empirically and extracted the following useful observations.

Differences in the architecture and the programming languages between the platforms require to handle the migration carefully. In contrast to Propeller that uses a lightweight C (due to limited memory - max memory is 32KB), Arduino supports full C/C++ commands. Consequently, software libraries that enable accessing components on Arduino (e.g., servo motor, DHT22 sensor) need additional declarations and supporting commands for object instantiation and initialisation, respectively. For instance, Arduino requires to define a servo variable globally (e.g., `Servo servoLeft()`) and perform extra tasks (in the `setup` method) to inform the servo object for the port that enables controlling the servo motor (e.g., `servoLeft.attach(leftSrvPrt)`). Using the servo motor on Propeller, however, needs no initialisation because the platform architecture expects that the left servo is attached to port #12. Likewise, instantiating the DHT22 sensor object on Arduino requires both the port and DHT model (e.g., `DHT(DHTPort, DHTModel)`) as the DHT library is used for multiple models, while Propeller does not need a sensor object.

The level of difficulty for inferring commands (i.e., invocations to Arduino libraries and auxiliary code) that achieve the same functionality as the original method depends on the correspondence between the platform libraries [8, 28]. When the correspondence is one-to-one, inferring the mappings is fairly simple. For instance, the commands `high(port)` and `digitalWrite(port, HIGH)` switch on an LED on Propeller and Arduino, respectively. In most cases, however, complex one-to-many and many-to-one mappings were needed. For example, the infrared transmitter on Propeller needs a single `freqout()` command to transmit the signal for a period of time, whereas on Arduino two commands should be used (i.e., `tone()` and `delay()`). Similarly, receiving a humidity reading on Propeller requires to trigger a reading from the DHT22 sensor and then to retrieve the humidity value, in contrast to a single command needed on Arduino.

Mappings inference is significantly more challenging in cases involving semantic differences between platform libraries. This discrepancy occurs for the libraries controlling the servos on the examined platforms. Moving the robot on Propeller requires a single invocation of the `drive_speed()` method (e.g, line 24 in Listing 1) that sets each servo to a certain speed in ticks per second. In contrast, the corresponding `servoLeft.write()` method from the Arduino servo library enables to control each servo individually by defining the exact pulse width in microseconds. Therefore, additional effort was needed to find suitable transformations in these situations.

RQ3 (Performance). We assessed the effect of the adapter pattern [13] to the size of software generated by RoboSMi and the software after completing the migration and found that they are on average 40% and 90%, respectively, larger than the software given as input. For the environmental monitoring application, for instance, the Propeller software is 1,506 bytes compared to 2,156 and 2,922 bytes for the generated software and that from the completed migration, respectively. Since RoboSMi generates placeholder methods for each Propeller-specific library method, a constant factor increase is expected. In the worst case, each line of the source software will make use of a Propeller library method, requiring three generated lines of code in the output software (method declaration; TODO directive; closing bracket). More lines may be added for generated port commands and target platform include directives. Without considering manual additions, the generated software has an upper bound $3L$ where L is the number of lines in the source software (cf. Listing 3). Furthermore, the size of the migrated software depends on the mappings between library methods for the source and target platforms. We found that achieving the same functionality using Arduino libraries requires more lines of code since component setup is generally more explicit than when using Propeller libraries (e.g. consider the servo and infrared transmitter examples discussed earlier). As such, software migrated directly to the target platform may be larger anyway without the use of the RoboSMi approach.

We did not identify any perceptible difference in performance when running the robotic applications on both source and target platforms. Other than platform differences, the only potential performance reduction could come from using additional method calls from the generated methods. However, compilers optimized for speed can inline functions to prevent this from occurring. Thus, the difference is likely to be negligible unless there is a cache miss. Other than this, performance differences are more likely to depend on the efficiency of the corresponding platform libraries and the hardware specifications for the platforms themselves.

RQ4 (Generality). While our evaluation targets software migration from Propeller C to C++, RoboSMi also supports migration of other combinations of procedural (C) and object-oriented (C++) programming paradigms. This entails that the source platform software (including platform-specific libraries) is in C/C++ and the migration is C→C, C→C++ or C++→C++. Specialising the adapter pattern to migrate from C++ to C, albeit feasible, needs advanced constructs (e.g., functions in C structures) that outweigh the benefits provided, and thus, manual migration or re-development may be preferred.

Beyond robotic systems, applying RoboSMi to other domains such as cyber-physical systems and Internet-of-Things requires small modifications. In particular, the analysis and library recommendation steps (Sections 3.1 and 3.3) require no changes subject to the source and target platform software combinations discussed above and the availability of a set of possible candidate libraries. Determining hardware components and communication ports for other Fritzing specifications is natively supported (Section 3.2), while supporting inputs form similar tools (e.g., KiCad, Eagle) requires small adaptations to the text-to-model transformation. Finally, the template for software generation (Section 3.4) should be adapted to match the structure expected by the target platform.

5.4 Threats to Validity

We mitigate **construct validity** threats that could be due to simplifications in the experimental setup by using robotic applications with complexity and size similar to applications in real-world

scenarios. We also employ hardware components widely-used in applications available on open-source robotic repositories (e.g., https://create.arduino.cc/projecthub).

We limit **internal validity** threats that could lead to bias in the process adopted for validating RoboSMi by assessing the correctness of each RoboSMi step individually through comparing the generated against the expected outputs. We reduce further these threats by manually completing the migration for both robotic applications and confirming their successful migration from a Propeller Activity board to an Arduino Uno.

We address **external validity** threats that could affect the generalisation of the RoboSMi tool-supported instance by using open-source and extensible software components widely-adopted in MDE. The source platform software and target platform hardware analysis steps are applicable to any robotic software written in C/C++ that could be analysed using Eclipse CDT [9] and to any hardware configuration provided as a Fritzing specification, respectively. The library recommendation step is generalisable to other applications while the TDIDF relevance score heuristic could be easily replaced with more sophisticated techniques. Some modifications might be needed for software generation for other target platforms. For example, the generated software currently conforms to the Arduino structure of `setup` and `loop` methods. Thus, our findings are not conclusive for all types of robotic software migration activities, and more experiments are needed to confirm the generality and scalability of the RoboSMi approach and tool.

6 RELATED WORK

One of the earliest projects focusing on model-based migration and modernisation was [12], which explored the use of reverse engineering, model transformation and model migration techniques to migrate a mainframe application to a more modern J2EE application. The novelty in this work was the development of a model-based process, which extracted a model from (legacy) source code. This model was then transformed into a platform-independent model (consisting of representations of static data structures, actions, application navigation logic, and user interfaces). The platform-independent model would then be transformed into a platform-specific (UML) model and thereafter to code. Some user intervention is typically required to ensure that the platform-independent model accurately captures all important information in the code. The approach, therefore, is not fully automated but nevertheless demonstrates improvements in the cost/effort in carrying out migration. There are also challenges in terms of supporting testing of the migrated application: test cases are not in general migrated with the application.

An OMG initiative to support standardization efforts in software migration was embodied in the *Architecture Driven Modernization (ADM)* initiative [25]. This led to the development of a set of standards to support reverse engineering, transformation, and migration activities. Of particular note is the *Knowledge Discovery Metamodel (KDM)* standard, which is a metamodel for representing the key knowledge pertaining to software assets in an enterprise system. It thus allows representation of the structure and behaviour of existing software systems at different levels of granularity (via its container concept). It provides facilities for representing software's operating environment, events and state transition behaviour, UI features, and persistent data. KDM is a representation standard, not an implementation, and does not provide mechanisms to support different software migration tasks.

The MoDisco framework [6] is an Eclipse project that has produced tools for modernising existing software systems, taking a pattern-based approach and using software models. It implements metamodels to describe existing systems, pattern-based discoverers for querying existing systems and gathering information needed to populate migrated models. The tools in MoDisco are themselves generic and support migration of documentation or code, while aiming to support different quality assurance processes. Thus, new discoverers can be built for new programming languages. It can also make use of KDM as a representation for the results of discoverers.

In [10], the authors demonstrate an approach, with a supporting toolchain, for software migration of the data layer of data-intensive applications to cloud infrastructure, based on the use of model transformations and the KDM. The approach is two-stage, and accurately estimates the migration cost, migration duration and cloud running costs of relational databases. The first stage obtains workload and structure models of the database to be migrated from database logs and the database schema. The second stage performs a discrete-event simulation using these models to obtain the cost and duration estimates. The approach focuses on cloud migration specifically, though it uses generic standards (e.g., KDM and other metamodels).

In [14], the authors demonstrate a purely code-based approach to software migration, exploiting Eclipse-based parsing and code generation tools. The approach aims to facilitate API, programming language, and hardware platform migration by analyzing source code, validating the source code to ensure that migration is feasible (e.g., by identifying code fragments that will require complex patterns to be imposed), refactoring the code base, and then generating target code. The approach was applied to several large code bases, and was shown to be useful to support profiling and hot-spot analysis. i.e., in identifying parts of the source application that may be challenging and expensive to migrate.

7 CONCLUSION

We presented RoboSMi, a model-driven approach to software migration of robotic applications between different hardware platforms. RoboSMi uses the software deployed on the source platform and a description of the architecture for the target platform to generate software that can run on the target platform and indicate areas that require manual adaptation by engineers. RoboSMi has been evaluated for the migration of two robotic applications from a Propeller Activity Board to an Arduino Uno. We found that RoboSMi can generate the correct artefacts for the software migration between hardware platforms and that the migrated applications are functionally equivalent with the original ones, although they are larger in terms of lines of code than the original because of using the adapter pattern. We plan to enhance RoboSMi with support for validating hardware architecture specifications using constraint languages and mapping inference techniques [28], and explore how to take into account the non-functional properties of the system. Finally, we plan to investigate possible RoboSMi extensions to support other programming language paradigms and assess its applicability to more complex and industrial-level robotic applications.

REFERENCES

[1] 2007. IEC 62402:2007 Obsolescence management. Application guide.
[2] 2007. JSPP 886, Volume 7, Part 8.13: Obsolescence management.
[3] Turki Alelyani, Ronald Michel, Ye Yang, Jon Wade, Dinesh Verma, and Martin Törngren. 2019. A literature review on obsolescence management in COTS-centric cyber physical systems. *Procedia computer science* 153 (2019), 135–145.
[4] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs Van Der Storm. 2009. Study of an API migration for two XML APIs. In *International Conference on Software Language Engineering*. Springer, 42–61.
[5] Marcel Bergerman, John Billingsley, John Reid, and Eldert van Henten. 2016. Robotics in agriculture and forestry. In *Springer handbook of robotics*. Springer, 1463–1492.
[6] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. Modisco: A model driven reverse engineering framework. *Information and Software Technology* 56, 8 (2014), 1012–1032.
[7] Mickael Clavreul, Olivier Barais, and Jean-Marc Jézéquel. 2010. Integrating legacy systems with mde. In *32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 69–78.
[8] Danny Dig and Ralph Johnson. 2005. The Role of Refactorings in API Evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 389–398.
[9] Eclipse Foundation. 2010. Eclipse CDT. https://www.eclipse.org/cdt. Accessed: 30-03-2019.
[10] Martyn Ellison, Radu Calinescu, and Richard F Paige. 2018. Evaluating cloud database migration options using workload models. *Journal of Cloud Computing* 7, 1 (2018), 6.
[11] Alessandro Farinelli, Elena Zanotto, Enrico Pagello, et al. 2017. Advanced approaches for multi-robot coordination in logistic scenarios. *Robotics and Autonomous Systems* 90 (2017), 34–44.
[12] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. 2007. Model-driven engineering for software migration in a large industrial context. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 482–497.
[13] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
[14] Simos Gerasimou, Maria Kechagia, Dimitris Kolovos, Richard Paige, and Georgios Gousios. 2018. On Software Modernisation Due to Library Obsolescence. In *2nd International Workshop on API Usage and Evolution (WAPI '18)*.
[15] Simos Gerasimou and Dimitris Kolovos. 2018. Epsilon CDT Driver. https://github.com/gerasimou/EMC-CDT. Accessed: 30-03-2019.
[16] Simos Gerasimou, Dimitris Kolovos, Richard Paige, and Michael Standish. 2017. Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems. In *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 385–393.
[17] Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrova, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. 2017. The STRANDS project: Long-term autonomy in everyday environments. *IEEE Robotics & Automation Magazine* 24, 3 (2017), 146–156.
[18] International Federation of Robotics. 2016. World Robotic Survey. (2016). https://ifr.org/news/world-robotics-survey-service-robots-are-conquering-the-world-/
[19] Javier Luis Cánovas Izquierdo and Jesús García Molina. 2014. Extracting models from source code in software modernization. *Software & Systems Modeling* 13, 2 (2014), 713–734.
[20] Connor Jennings, Dazhong Wu, and Janis Terpenny. 2016. Forecasting obsolescence risk and product life cycle with machine learning. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 6, 9 (2016), 1428–1439.
[21] André Knörig, Reto Wettach, and Jonathan Cohen. 2009. Fritzing: a tool for advancing electronic prototyping for designers. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. ACM, 351–358.
[22] Dimitrios Kolovos, Richard F Paige, and Fiona AC Polack. 2006. The Epsilon object language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 128–142.
[23] Dimitrios Kolovos, Louis Rose, Richard Paige, and Antonio Garcıa-Domınguez. 2010. The Epsilon book. *Structure* 178 (2010), 1–10.
[24] Lloyd's Register Foundation. 2016. Foresight review of robotics and autonomous systems. (2016). https://www.lrfoundation.org.uk/en/news/foresight-review-of-robotics-and-autonomous-systems
[25] Object Management Group, Inc. 2012. Architecture-Driven Modernization Task Force. (2012). http://adm.omg.org/
[26] Claudia Raibulet, Francesca Arcelli Fontana, and Marco Zanoni. 2017. Model-driven reverse engineering approaches: A systematic literature review. *IEEE Access* 5 (2017), 14516–14542.
[27] S Rajagopal, JA Erkoyuncu, and R Roy. 2015. Impact of software obsolescence in defence manufacturing sectors. *Procedia CIRP* 28 (2015), 197–201.
[28] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637.
[29] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. 2008. The Epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 1–16.
[30] Gerard Salton and Michael J McGill. 1986. Introduction to modern information retrieval. (1986).
[31] Peter Sandborn and Jessica Myers. 2008. *Designing Engineering Systems for Sustainability*. 81–103.
[32] Bruno Siciliano and Oussama Khatib. 2016. *Springer handbook of robotics*. Springer.
[33] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lammel. 2010. Swing to SWT and back: Patterns for API migration by wrapping. In *International Conference on Software Maintenance*. IEEE, 1–10.
[34] Xi Vincent Wang, Lihui Wang, Abdullah Mohammed, and Mohammad Givehchi. 2017. Ubiquitous manufacturing system based on Cloud: A robotics application. *Robotics and Computer-Integrated Manufacturing* 45 (2017), 116–125.
[35] John-David Warren, Josh Adams, and Harald Molle. 2011. *Arduino for Robotics*. Apress, 51–82.